Interni kurs

RT-RK
COMPUTER BASED SYSTEMS

Advanced C++ programming

Predavač:
Goran Jakovljević, goran.jakovljevic@rt-rk.com

Beograd, 9.12. – 13.12.2019.

O kursu

- C++11
- C++14
- C++17
- Uvedeno mnogo raznih novina.
- Cilj ovog kursa je da ukaže na najvažnije (mada je to većina tih novina).

Literatura

- Biblioteka RT-RK:
- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, Scott Meyers, O`Reilly, 2014
- Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs, Scott Meyers, Addison-Wesly, 2005.

Pregled tema

- Odabrana jednostavnija proširenja jezika
- Move semantika
- Pametni pokazivači
- Lambda funkcije
- Algoritmi standardne biblioteke
- Generičko programiranje
- Višenitno programiranje

RT-RK
COMPUTER BASED SYSTEMS

Odabrana jednostavnija proširenja jezika

nullptr

- Koje vrednosti pokazivač može da ima?
- Vrednost možemo videti:

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

- Ali nam je to jako retko potrebno (suštinski nikad, osim za neka ozbiljna debagovanja).
- Međutim, potrebno nam je da razlikujemo dva slučaja:
  - pokazivač pokazuje na neki objekat
  - pokazivač ne pokazuje ni na šta

nullptr

- Od svih mogućih vrednosti (adresa) koje pokazivač može da ima, potrebno je da jedna bude specijalna – rezervisana da znači „ni na šta ne pokazujem“. Ta vrednost se naziva „Null vrednost“.
- Standard ne specificira koja to brojčana vrednost treba da bude.
- Da bismo razlučili između pokazivača koji pokazuje na neki objekat i pokazivača koji ne pokazuje ni na šta, moramo imati neku oznaku za tu vrednost.
- U praksi se za to koristio (do C++11) znak: 0.

nullptr

- U zavisnosti od konteksta, 0 se interpretira kao ceo broj 0, ili kao vrednost pokazivača ni na šta (na toj konkretnoj platformi)
- To rešenje je donelo bar dva problema:
  - Logički se meša ceo broj nula sa Null vrednošću (koja nije celobrojnog tipa, niti mora biti brojčano jednaka nuli)
  - U određenim kontekstima upotreba dovodi do višeznačnosti

```
void foo(int x);
void foo(char *x);
foo(0); // na šta se ovde misli?
```

nullptr

- Kao polurešenje se opet (opet pre C++11) nametnula upotreba pretprocesorskog simbola **NULL**, koji je po standardu definisan u zaglavlju **stddef.h** (odnosno **<cstddef>**).
- U C++11 je uvedena nova rezervisana reč da označava Null verdnost: **nullptr**, što konačno predstavlja potpuno rešenje.
- Upotreba znaka 0 je i dalje moguća, zbog kompatibilnosti sa postojećim programima, ali se njena upotreba obeshrabruje.

```
void foo(int x);
void foo(char *x);
foo(nullptr); // sada je stvar jasna
```

auto

- Kojeg tipa treba da su ove promenljive, ako ove inicijalizacije treba da budu valjane i bez implicitnih konverzija?

```
a = 5;
b = 5.0;
c = 1.0 + 2.0i;
d = new Struct();
e = v.begin();
```

auto

- Kojeg tipa treba da su ove promenljive, ako ove inicijalizacije treba da budu valjane i bez implicitnih konverzija?

```
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** ključna reč sada znači da tip promenljive treba da odgovara tipu inicijalizacionog izraza

```
auto a = 5;
auto b = 5.0;
auto c = 1.0 + 2.0i;
auto d = new Struct();
auto e = v.begin();
```

auto

- **auto** nije nova ključna reč, ali ima novu svrhu.
- Šta je do C++11 **auto** značilo i zašto smo ga retko viđali (praktično nikada)?
- **auto** ne treba sejati gde god možete (AAA – Almost Always Auto).
- Ta ključna reč **ne znači** da se ne mora znati tip promenljive, ili da se o tome ne mora misliti.
- Glavni razlozi za upotrebu auto su:
  - u slučajevima složenog tipa
  - u implementaciji i upotrebi šablona (generičko programiranje)
  - kod lambda funkcija
  - održavanje koda

auto

- Ne radi dobro uz uniformnu inicijalizaciju – korišćenje {}.
- Ponekad auto proizvodi neželjen tip.

```
std::vector<bool> status(Elem &W);
bool color = status(W)[5];
process(W, color);

std::vector<bool> status(Elem &W);
auto color = status(W)[5]; //-> std::vector<bool>::reference
process(W, color);

Matrix M = m1 + m2 + m3;
auto M1 = m1 + m2 + m3;
Sum<Sum<Matrix, Matrix>, Matrix> M1 = m1 + m2 + m3;
```

range for loop

- **for** petlja sada ima dodatan oblik:

```
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;
}
```

range for loop

- **for** petlja sad ima dodatan oblik:

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

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

range for loop

- Radi i za C-ovski niz:

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

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

range for loop

- Da bi radilo za korisničke tipove (a svi STL kontenjeri su korisnički tipovi), potrebno je da korisnički tip ima iteratore i metode begin() i end().
- Iterator je tip koji ima definisane sledeće operacije:
  - == (i !=)
  - *
  - ++
- begin() i end() vraćaju iterator na početak, odnosno iterator iza kraja kontenjera.
- ali, mogu biti begin i end funkcije:

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

Atributi

- Da li ste videli ovo u C/C++ kod koji se prevodi GCC-om?

```
__attribute__((nešto))
```

- Ili ovako nešo:

```
#pragma nešo
```

- Moderni C++ nudi novu sintaksu koja bi trebalo da bude standardizovana zamena za __attribute__ (i druga kompajlerska proširenja te namene) i alternativa pretprocerskoj konstrukciji #define
- Ovako izgleda standardni C++ atribut:

```
[[nešto]]
```

- Može da stoji uz tipove, promenljive, funkcije, blokove koda i naredbe, pa ča i da važi i za cele jedinice prevođenja.

Atributi

- Standard definiše sledeće atribute:

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

- Standard takođe propisuje i ovo pravilo kako bi podržao atribute koji su specifični za platformu: Ako kompajler ne prepoznaje neki atribut, treba da ga ignoriše (ni upozorenje ne treba da se prijavi).

Atributi

- Kada funkcija neće vratiti kontrolu pozivajućoj funkciji.

```
[[noreturn]] void foo() {
  throw “error”;
}
[[noreturn]] void bar() {
  while (true);
}
```

- Pomaže kompajleru da optimizuje u nekim slučajevima, plus eliminiše neka upozorenja koje bi možda kompajler generisao.

Atributi

- Kada je neki element programa/biblioteke zastareo i trebalo bi ga polako napustiti.

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

[[deprecated]] class SomeClass;

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

- A može da stoji uz enume, definicije tipova, promenljive...

Atributi

- Može i da se naznači da je propadanje kroz drugu case labelu namerno:

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

Atributi

- Kod nekih funkcija želimo da osiguramo da će povratna vrednost biti uvažena:

```
[[nodiscard]] bool sanityCheck();
void foo() {
   sanityCheck();
   // ...
}
```

- A nekad je korisno naznačiti da se neka promenljiva možda neće uvek koristiti (pa da kompajler to ne prijavljuje kao upozorenje)

```
void foo() {
   [[maybe_unused]] bool sanityOK = sanityCheck();
   assert(sanityOK); // neće biti pristupno u release build-u
   // ...
}
```

Enum class

- Postoje tri problemčića sa dosadašnjim nabrojivim tipovima:

1. Definisani simboli upadaju u trenutni doseg.

```
enum PeriniDrugari { SIMA, DJURA, STEVA };
enum MikiniDrugari { DJOLE, SIMA, MILE };
SIMA // na šta se ovde misli?
```

2. Celobrojni tip na koji se enum svodi je vrlo labavo definisan
   - Pa se na to ne može osloniti, a posledice su da nije pouzdano raditi aritmetiku sa enum vrednostima, niti je moguće samo deklarisanje enuma u napred.
3. Enum vrednosti se implicitno konvertuju u ceo broj.
   - U kombinaciji sa prethodnim olakšava da se slučajno napravi greška.

Enum class

- Ti problemi su se do sada ovako rešavali:

1. Definisani simboli upadaju u trenutni doseg.

```
enum PeriniDrugari { PD_SIMA, PD_DJURA, PD_STEVA };
enum MikiniDrugari { MD_DJOLE, MD_SIMA, MD_MILE };
PD_SIMA // Sada je jasno.
```

2. Celobrojni tip na koji se enum svodi je vrlo labavo definisan
   - Treba pažljivo pisati kod da se na to ni ne oslanja. Ukratko: koristi enume samo za skladištenje vrednosti i poređenje i to samo sa simbolima iz istog to enuma.
3. Enum vrednosti se implicitno konvertuju u ceo broj.
   - Prosto paziti malo više. Neki kompajleri prijavljuju upozorenje.

Enum class

- Sad imamo nabrojivu klasu:

1. Pravi svoj doseg.

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

2. Svodi se uvek na int, osim ako eksplicitno nije navedeno drugačije.
   - Kod deklaracije unapred, podrazumeva se int. Ukoliko je u trenutku definicije neka od vrednosti van opsega, kompajler će prijaviti grešku.

```
enum class Primer1 { A, B, C }; // svodi uvek na int
enum class Primer2 : long { X, Y, Z }; // sada na long
enum class Primer3;
…
enum class Primer3 { P, Q, R };
```

3. Vrednost iz enum klase se ne konvertuje implicitno u ceo broj.
   - Ako treba, mogu se eksplicitno konvertovati pomoću static_cast.

const

- const u C++-u, u suštini ima značenje „samo za čitanje“ („read-only“)

```
int g_a;
void foo() {
  const int k = g_a;
  k += 1; // greška u prevođenju
  g_a += 1; // OK
}
void bar(const int& x);
const volatile int* x = FLAG_ADDRESS;
```

Kompajliranje nasuprot interpretaciji

- Kod kompajliranja faza prevođenja je vremenski odvojena od faze izvršavanja.

- Kod interpretacije, te dve faze su vremenski isprepletane.

const

- **const** govori kompajleru da tokom prevođenja prijavi kao grešku neredbe koje bi tokom izvršavanja izmenile (ili mogle da izmene) to što je obeleženo kao const. Zbog toga još kažemo i da je to „konstantno tokom izvršavanja“ (eng. „run-time constant“).

- Kompajler u opštem slučaju ne zna koja će to konkretno vrednost biti (niti ga to zanima, osim u nekoliko specifičnih slučajeva kada pokušava da optimizuje kod), jedino mu je bitno da se samo jednom inicijalizuje (na početku dela gde se koristi) i da se posle ne menja tokom izvršavanja.

const

- Ovo su sve primeri gde kompajer nikako ne može da znati vrednost tokom prevođenja.

```
int g_a;
void foo() {
  const int k = g_a;
  k += 1; // greška u prevođenju
  g_a += 1; // OK
}
void bar(const int& x);
const volatile int* x = FLAG_ADDRESS;
```

const

- Ali, ovo su primeri gde može:

```
const int g_a = 5;
void foo() {
  const int k = g_a;
  const int j = 7;
  std::cout << k << j << std::endl;
}
```

- I kompajer će u većini slučajeva gornji kod svesti na donji:

```
const int g_a = 5;
void foo() {
  std::cout << 5 << 7 << std::endl;
}
```

- U ovim slučajevima možemo govoriti o „konstantama tokom prevođenja“, ali to je sa stanovništa jezika samo pitanje optimizacije.

constexpr

- Sad postoji nova reč koja označava da nešto baš treba da bude „konstantno tokom prevođenja“ (eng. „compile-time constant“).
- Dakle, kompajler tada mora znati vrednost toga tokom prevođenja.

```
constexpr int g_a = 5;
void foo() {
  constexpr int k = g_a;
  constexpr int j = 7;
  std::cout << k << j << std::endl;
}
```

- Ovakva deklaracija i inicijalizacija promenljive predstavlja pandan makroima s tim što constexpr omogućava sledeće pogodnosti:
  - provera ispravnosti izraza sa constexpr proneljivom na osnovu njenog tipa
  - preglednije i lakše predstavljanje kompleksnih konstanti tokom prevođenja.

constexpr

- Kompajler će se sad buniti ako vrednost ne može da zna tokom prevođenja

```
consexpr volatile int* x = FLAG_ADDRESS; // greška u
                                         // prevođenju
int g_a;
void foo() {
  consexpr int k = g_a; // greška u prevođenju
}

void bar (constexpr int& x); // greška u prevođenju
```

- Gde god nam je namera da bude konstanta tokom prevođenja, treba da upotrebimo constexpr. Jasnije izražavamo nameru, kompajler proverava da li smo postigli to što želimo, a ujedno i osiguravamo optimizaciju po tom pitanju.

constexpr

- Ako upotrebimo constexpr, onda je sračunavanje tog izraza tokm prevođenja obaveno.

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

- Sada možemo primetiti i da je cela povratna vrednost funkcije konstantna tokom prevođenja.

```
constexpr int foo() {
  return 6 * 5 + 2;
}
constexpr int a = foo(); // Skroz OK.
                         // Kao da je a = 32;
```

constexpr funkcije

- constexpr funkcije mogu da primaju parameter.

```
constexpr int foo(int x) {
  return 5 * 6 + x;
}
constexpr a = foo(2); // Skroz OK. Kao da je a = 32;
constexpr b = foo(a); // b = 62; (5 * 6 + 32)
```

- Ali može i ovo:

```
void bar(int y) {
  int a = foo(y); // I dalje u redu, ali neće biti sračunato tokom
                  // prevođenja, već će biti regularan poziv funkcije.
}
```

- Dakle, constexpr funkcije moraju biti sračunljive tokom prevođenja (konstantne tokom prevođenja) ako su im stvarni parametri sračunljivi tokom prevođenja. U suprotnom, biće generisan kod za njih kao za redovne funkcije.

constexpr funkcije

- Standard propisuje da constexpr funkcija mora poštovati oderđena ograničenja da bi se osigurala njena sračunljivost tokom prevođenja (u suprotnom, kompajler je neće prihvatiti kao constexpr funkciju).
- Otprilike, njen kod mora biti takav da nema sporednih efekata (uticaja na nešto van funkcije).
- Konkretnije, u telu funkcije se smeju koristiti svi iskazi i izrazi osim:
  - asemblerskog izraza
  - goto
  - labela (osim case i default)
  - try blok
  - poziva funkcija koje nisu constexpr
  - definicija promenljivih bez inicijalizacije
  - definicija promenljivih statičke trajnosti (ili nitske tajnosti – o tome kasnije)
  - definicija promenljivih neliteralskog tipa (videćemo kasnije, ali ukratko: u obzir dolaze samo jednostavni, plitki tipovi, sa trivijalnim destruktorom i constexpr konstruktorom; npr. std::string ne može).

constexpr funkcije

- Između C++11 i C++14 standarda, bilo je restriktivnije.
- constexpr funkcija se mogla sastojati samo od jedne return naredbe.
- Za if se može koristiti ? :
- I rekurzijom se može oponašati petlja.
- Od C++14 void se smatra literalskim tipom i menjanje constexpr objekata je dozvoljeno samo constexpr metodama.

constexpr promenljive

- constexpr promenljiva mora biti literalskog tipa.
- Neformalno rečeno, to su jednostavni, plitki tipovi, koji mogu biti konstruisani (inicijalizovani) prilikom interpretacije od strane prevodioca, i sa kojim se račun može obaviti tokom prevođenja.

constexpr promenljive

- Inicijalizacioni izraz za constexpr promenljive može da se sastoji od sledećih elemenata:
  - literala
  - drugih constexpr promenljivih
  - poziva constexpr funkcija (operacije nad osnovnim tipovima su za poterbe ove definicije constexpr funkcije)
  - i const promenljivih celobrojnog ili nabrojivog (enum) tipa koje su po svojoj prirodi konstante tokom prevođenja.
- Ova poslednja stavka omogućava da se constexpr doda u stari kod, bez temeljnog prepravljanja.

```
const int g_a = 5;
const duble g_b = 5.0;
constexpr double g_c = 6.0;
void bar() {
  constexpr int k = g_a;
  constexpr double j = g_b; // ovo ne može
  double i = g_c; // ali ovo može
}
```

Statičke tvrdnje

- Još od jezika C imamo mehanizam tvrdnji (assert), definisan u zaglavlju assert.h (#include <cassert> u C++-u).

```
// Prima parne brojeve. U suprotnom – nedefinisano stanje.
int foo(int x) {
  assert(x % 2 == 0);
  ...
}
```

- U debug build-u na mestu tvrdnje naći će se kod koji se proverava tokom izvršavanja. Ukoliko tvrdnja nije zadovoljena, izvršavanje programa će se prekinuti uz prijavu greške.
- U release build-u tvrdnje nestaju i tokom izvršavanja se ne obavlja nikakva provera.

Statičke tvrdnje

- U C++11 uveden je mehanizam za statičke tvrdnje (static_assert).
- To su tvrdnje koje mogu biti proverene tokom prevodjenja.

```
static_assert(sizeof(int) * CHAR_BIT == 32,
              “int has to be 32 bits for this code to work”);
// a može i bez poruke o grešci:
static_assert(sizeof(int) * CHAR_BIT == 32); // biće generisana neka
                                             // opšta poruka.
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”);
```

- Pravi parametar statičke tvrdnje mora biti izraz sračunljiv tokom prevođenja.

Statičke tvrdnje

- Izraz u statičkoj tvrdnji se tokom prevođenja sračunava samo jednom, u jednom kontekstu.
- Zato statičke tvrdnje **ne možemo** iskoristiti za proveru ulaznih parametara ni u constexpr funkcijama.

```
// Prima parne brojeve. U suprotnom – nedefinisano stanje.
constexpr int foo (int x) {
  static_assert(x % 2 == 0); // GREŠKA!
  // Izraz nije sračunljiv, zbog x koje zavisi od poziva
  // do poziva, a static_assert se sračunava samo jednom
  ...
}
```

Statičke tvrdnje

- Ali, obične tvrdnje sasvim lepo rade u constexpr funkcijama:

```
// Prima parne brojeve. U suprotnom – nedefinisano stanje.
constexpr int foo(int x) {
  assert(x % 2 == 0);
  ...
}
constexpr int a = foo(4); // OK
constexpr int b = foo(5); // Greška tokom prevođenja
```

Statičke tvrdnje

- Takođe, constexpr funkcija može da baci izuzetak:

```
// Prima parne brojeve. U suprotnom – nedefinisano stanje.
constexpr int foo(int x) {
  if (x % 2 != 0) throw std::logic_error("nije parno");
  ...
}
constexpr int a = foo(4); // OK
constexpr int b = foo(5); // Greška tokom prevođenja
// a сада може и ово:
 void bar(int v) {
   try {
     int r = foo(v); // prevodi se kao regularna funkcija
   }
   catch (...) {//...}
}
```

noexcept

- Problemi sa starom dinamičkom specifikacijom izuzetaka.

```
void foo(int x) throw(lista izuzetaka);
```

- Pozivalac funkcije nije u obavezi da „uvaži“ ovu specifikaciju.
- Kada se koriste šabloni, često je nemoguće predvideti tip izuzetka.
- U praksi jedinu upotrebu imala je prazna lista izuzetaka.

```
void foo(int x) throw();
```

noexcept

- Stara dinamička specifikacija izuzetaka je označena kao zastarela.
- Uvedena je nova ključna reč: **noexcept**

```
void foo(int x) noexcept;
```

- Zamena za throw(), ali sa unapređenim performansama.
  - Kao i u starom slučaju, ako izuzetak izađe iz ovakve funkcije, program se prekida.
  - Razlika je što stek ne mora da se razmotava, a to omogućava kompajleru da napravi efikasniji kod.
- Specifikacija izuzetaka (bez reči dinamička) se odnosi na novu specifikaciju.
- Od C++17 je i deo interfejsa funkcije.

noexcept

- Uslovna specifikacija

```
noexcept(true); // je isto što i noexcept;
```

- Takođe može da zavisi od statusa operacija koje se koriste u toj funkciji:

```
template<class T1, class T2>
class pair {
  // ...
  void swap(pair& __p)
    noexcept(noexcept(swap(first, __p.first))
             && noexcept(swap(second, __p.second)))
  {
    swap(first, __p.first);
    swap(second, __p.second);
  }
  // ...
  T1 first;
  T2 second;
}
```

noexcept

- Kako noexcept može biti uslovan izraz u nekim funkcijama i kako je on deo interfejsa funkcije treba ga dobro i dugoročno planirati.
- Treba imati u vidu da je većina funkcija neutralna po pitanju izuzetka (eng. exception neutral). To znači da ona sadrži pozive funkcija koje mogu baciti izuzetak ali ne upravlja njima već ih prosleđuje dalje u lancu poziva funkcija. Te funkcije se ne smeju deklarisati kao noexcept.

noexcept

- Sledeće metode su implicitno označene kao noexcept:
  - Destruktor
  - Razne forme operatora delete ()
- Mogu se eksplicitno prepraviti da budu noexcept(false), ali nije dobra ideja.
- Move operacije (sledeće predavanje) je vrlo korisno označiti kao noexcept.

Dve novine sa string literalima

- Šta je tip ovog izraza:

```
"...prosta recenica?"
```

Dve novine sa string literalima

- Šta je tip ovog izraza:

```
"...prosta recenica?"
```

- Do nedavno je bio **char***
- Zašto ne **const char***?
- Iz istorijskih razloga: prve verzije jezika C nisu imale const ključnu reč ali jesu imale string literale, i eto...
- Bilo kako bilo, sad je to ispravljeno i tip gornjeg izaza je sada **const char***, kao što je trebalo da bude. I očekivano je da kompajer prijavi grešku ukoliko to vaš kod ne uvažava.
- Svakako je se dovodilo do nedefinisanog stanja ako se pokuša upisati u memoriju koja je dodeljena string literalu!

Dve novine sa string literalima

- Ako želimo od ovakvog teksta da napravimo string literal:

```
U nazivu datoteke ne smeju da se koriste “\” i “’”
```

- Morali bi da pišemo ovako:

```
“U nazivu datoteke ne smeju da se koriste \“\\\” i \“\’\””
```

- A sada možemo i ovako:

```
R“(U nazivu datoteke ne smeju da se koriste “\” i “’”)”
```

- A ako baš želimo možemo i ovo da imamo u stringu:)“

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

- Na osnovu toga „a“ između “ i zagrade, zna se šta je zapravo kraj.

Dedukcija tipova

- C++98 koristi dedukciju tipova samo za šablone (templejte).
- C++11 koristi dedukciju tipova za auto i decltype koja za bazu ima dedukciju šablona.

```
template<typename T>
void f(ParamType param);

f(expr);
```

- U osnovi zaključivanja tipova šablonske funkcije dedukujemo dva tipa:

```
template<typename T>
void f(const T& param);

int x = 0;
f(x);
```

- U ovom primeru T je int dok je je ParamType dedukovan na const int&.

Dedukcija tipova šablona

- Kako dedukcija tipa T zavisi od forme ParamType tako imamo sledeća tri slučaja zaključivanja:
  - ParamType je pokazivač ili referenca, ali ne univerzalna referenca
  - ParamType je univerzalna referenca
  - ParamType nije ništa od navedenog

Dedukcija referentnog ili pokazivačkog tipa

- Pravilo nalaže da se referenca ili pokazivač tipa izraza expr koji inicijalizuje šablon ignoriše, a zatim da se expr usaglasi sa ParamType kako bi se na kraju odredilo T.

```
template<typename T>
void f(T& param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T je int, ParamType je int&
f(cx); // T je const int, ParamType je const int&
f(rx); // T je const int, ParamType je const int&
```

Dedukcija referentnog ili pokazivačkog tipa

```
template<typename T>
void f(const T& param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T je int, ParamType je const int&
f(cx); // T je const int, ParamType je const int&
f(rx); // T je const int, ParamType je const int&
```

Dedukcija univerzalne reference

- Univerzalne reference prihvataju i desne i leve vrednosti i uvek su istog oblika (T&&)
- Kod univerzalnih referenci ako je prosleđeni izraz expr leva vrednost T i ParamType će biti leve vrednosti, u suprotnom prethodna pravila se primenjuju.

```
template<typename T>
void f(T&& param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // x je l-vrednost pa je T tipa int&, ParamType je int&
f(cx); // cx je l-vrednost pa je T tipa const int&, ParamType
       // je const int&
f(rx); // cx je l-vrednost pa je T tipa const int&, ParamType
       // je const int&
f(27); // 27 je d-vrednost pa je T tipa int, ParamType je int&&
```

ParamType nije ni referenca ni pokazivač

- U ovakvom slučaju prosleđujemo po vrednosti.
- Ako je inicijalizujući izraz referenca ona se ignoriše (ali se pokazivač ne ignoriše).
- Ako expr izraz sadrži const ili volatile ignorisati ih.

```
template<typename T>
void f(T param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);   // T je int
f(cx);  // T je int
f(rx);  // T je int
```

Auto dedukcija

- auto dedukcija je šablonska dedukcija

```
auto x = 27;
const auto cx = x;
const auto& rx = x;
```

- Gornje deo koda se može preslikati u sledeće šablonske funkcije:

```
template<typename T>
void func_for_x(T param);
func_for_x(27);

template<typename T>
void func_for_cx(const T param);
func_for_cx(x);

template<typename T>
void func_for_rx(const T& param);
func_for_rx(x);
```

Auto dedukcija

- Međutim za auto dedukciju postoji još jedno specifično pravilo po kojem se razlikuje od dedukcije šablona, a ono je vezano za inicijalizaciju vitičastim zagradama.

```
int x1 = 27;     auto x1 = 27;
int x2(27);      auto x2(27);
int x3 = {27};   auto x3 = {27}; // x3 je std::initializer_list<int>
int x4{27};      auto x4{27};    // x4 je std::initializer_list<int>
```

- Šta se dešava u sledećem slučaju?

```
auto x5 = {1, 2, 3.0};
```

- A u ovom?

```
template<typename T>
void f(T param);
f({11,23,9});
```

Auto dedukcija

- Međutim za auto dedukciju postoji još jedno specifično pravilo po kojem se razlikuje od dedukcije šablona, a ono je vezano za inicijalizaciju vitičastim zagradama.

```
int x1 = 27;      auto x1 = 27;
int x2(27);       auto x2(27);
int x3 = {27};    auto x3 = {27}; // x3 je std::initializer_list<int>
int x4{27};       auto x4{27};    // x4 je std::initializer_list<int>
```

- Šta se dešava u sledećem slučaju?

```
auto x5 = {1, 2, 3.0};
```

- Rešenje je da promenimo ParamType

```
template<typename T>
void f(std::initializer_list<T> param);
f({11,23,9});
```

decltype

- Ukoliko se ime ili neki izraz prosledi kao argument u declype on vraća tip tog imena ili izraza.

```
const int i = 0;          // decltype(i) je const int
bool foo(const int& x); // decltype(foo) je bool(const int&)

struct MojTip {
  int a,b;                // decltype(MojTip::a) je int
};
MojTip t;
if (f(t))...              // decltype(f(t)) je bool

std::vector<int> v;    // decltype(v) je std::vector<int>
if (v[0] == 0)...      // decltype(v[0]) je int&
```

decltype

- Glavna upotreba decltype-a je prilikom zaključivanja povratne vrednosti šablonske funkcije.

```
template<typename C, typename I>
auto getElem(C& c, I i) -> decltype(c[i])
{
   ...
   return c[i];
}

std::deque<int> d;
getElem(d, 5) = 10; // OK jer je getElem(d, 5) tipa int&
```

- auto pre imena funkcije u C++11 znači da će se tip povratne vrednosti zaključiti na osnovu implicitno zadatog povratnog tipa nakon potpisa funkcije.

decltype

- U C++14 auto se može koristiti i bez implicitnog postavljanja povratnog tipa funkcije. Tom prilikom se koriste pravila za zaključivanje šablonskih funkcija.

```
template<typename C, typename I>
auto get(C& c, I i)
{
   ...
   return c[i];
}
```

- Koji je sad tip povratne vrednosti?

decltype

- U C++14 auto se može koristiti i bez implicitnog postavljanja povratnog tipa funkcije. Tom prilikom se koriste pravila za zaključivanje šablonskih funkcija.

```
template<typename C, typename I>
auto get(C& c, I i)
{
   ...
   return c[i];
}
```

- Koji je sad tip povratne vrednosti?

```
std::deque<int> d;
getElem(d, 5) = 10; // GREŠKA jer je getElem(d, 5) tipa int
```

decltype

- U prethodnom primeru smo koristili pravila zaključivanja šablononskih funkcija. A šta se dešava ako hoćemo da koristimo decltype pravila zaključivanja?

```
template<typename C, typename I>
decltype(auto) getElem(C& c, I i)
{
  ...
  return c[i];
}
```

- Definisana povratna vrednost funkcije govori da prilikom zaključivanja povratne vrednosti funkcije želimo da koristimo decltype pravila zaključivanja.
- Isto pravilo možemo koristi i za inicijalizaciju promenljivih:

```
int x;
const int& cx = x;
auto y = cx;              // y ima tip int
declype(auto) z = cx; // z ima tip const int&
```

override

- Za preklapanje virtualne funkcije potrebno je
  - **deklaracija virtualne funkcije**
  - da ime bude isto
  - da tip funkcije bude isti

```
class B {
      void f1();
      virtual void f2(char);
      virtual void f3(char) const;
      virtual void f4(int);
};
```

```
class D : public B {
  void f1();       // ne preklapa
  void f2(int);    // ne preklapa
  void f3(char);   // ne preklapa
  void f4(int);    // preklapa
};
```

override

```
class B {
  virtual void f2(char);
  virtual void f3(char) const;
  virtual void f4(int);
};


class D : public B {
  void f2(int);  // ne preklapa i ne prijavljuje grešku
  void f3(char); // ne preklapa i ne prijavljuje grešku
  void f4(int);  // preklapa
};
```

override

```
class B {
  virtual void f2(char);
  virtual void f3(char) const;
  virtual void f4(int);
};


class D : public B {
  void f2(int) override;  // prijavljuje grešku
  void f3(char) override; // prijavljuje grešku
  void f4(int) override;  // preklapa i dalje
};
```

final

```
class B {
  virtual void f4(int);
};
class D : public B {
  void f4(int) override;
};
class D1 : public D {
   void f4(int) final;
};
class D2 : public D1 {
  void f4(int) override; // greška
};
```

final

Može i nasleđena klasa da bude final. To znači da nema daljeg nasleđivanja.

```
class B {
  virtual void f4(int);
};
class D : public B {
  void f4(int) override;
};
class D1 final : public D {
  void f4(int) override;
};
class D2 : public D1 { // ovde je sad greška
  ...
};
```

explicit

- Zamislimo da pravimo svoj tip realnog broja:

```
class MyReal {
  MyReal(double x);
  operator double() const;
};

void foo(MyReal x);
void bar(double x);

int main() {
  MyReal a = 5.0;
  foo(6.0);
  bar(a);
  a = a + 7.0;
  a = a + a; // čak će i ovo da radi
}
```

explicit

```
class MyReal {
  MyReal(double x);
  operator double() const;
  friend MyReal operator+(MyReal x, MyReal y);
};

void foo(MyReal x);
void bar(double x);

int main() {
  MyReal a = 5.0;
  foo(6.0);
  bar(a);
  a = a + 7.0; // ovo se sad neće prevesti!
  a = a + a;   // šta će ovde biti?
}
```

explicit

```
class MyReal {
  explicit MyReal(double x);
  explicit operator double() const;
  friend MyReal operator+(MyReal x, MyReal y);
};

void foo(MyReal x);
void bar(double x);

int main() {
  MyReal a = 5.0; // ne prevodi se
  foo(6.0);       // ne prevodi se
  bar(a);         // ne prevodi se
  a = a + 7.0;    // ne prevodi se
  a = a + a;      // ali ovde je sad stvar vrlo jasna
}
```

explicit

```
class MyReal {
  explicit MyReal(double x);
  explicit operator double() const;
  friend MyReal operator+(MyReal x, MyReal y);
  friend MyReal operator+(MyReal x, double y);
};

void foo(MyReal x);
void bar(double x);

int main() {
  MyReal a{5.0};     // ali ovako može
  foo(MyReal(6.0)); // ili foo(static_cast<MyReal>(6.0))
  bar(double(a));    // ili bar(static_cast<double>(a))
  a = a + 7.0;       // sada se prevodi
  a = a + a;         // i dalje OK
}
```

bool kao mali izuzetak

```
struct MyReal {
  explicit operator bool() const;
  …
};
void foo(bool x);
int main() {
  MyReal a;
  ...
  foo(a);  // ovo ne može
  if (a) { // ali ovo može
    ...
  }
}
```

Literali

- Za ugrađene tipove postoje literali (neposredni operandi)
- Kod aritmetičkih literala, tip je određen sufiksom i postojanjem tačke:

2U	-37
3L	31
31	051 (41)
0x2b (43)	0xFFFFFFD1 (-47)
11ULL	13.0
17.	3.14159
19.0F	6.02e32 (6.02 x 10^23)
23.F	1.6e-19 1.6 x 10^-19
29.0L	31.L
'c' (ovo je ceo broj tipa char)	

- A postoje i string literali:

```
"c"       // kog je ovo tipa?
"Pera"
"Marko"s // a ovo? Od C++14
```

Korisnički literali

- Ali, sada možemo i sami praviti literale.

```
class MyReal {
  ...
};
MyReal operator"" _mr(long double x);
void main() {
  std::cout<< 9.0_mr;
}
```

- Broj do sufiksa će biti tumačen kao ***long double***, i tako će biti prosleđen funkciji ***operator"" _mr***.
- Sufiks mora počinjati sa _.
- Ovo nije dobar literal, u skladu sa gornjim kodom.

```
9_mr
```

Korisnički literali

- Na raspolaganju imamo sledeće funkcije:

```
MyReal operator""_mr(long double x);
    // Hvata ovo: 9.0_mr, .5_mr, 1.6e-19_mr
MyReal operator""_mr(unsigned long long x);
    // Hvata ovo: 9_mr, 0x6_mr, 0b1010_mr, 076_mr
MyReal operator""_mr(char x);
    // Hvata ovo: 'a'_mr, 'B'_mr, 'v'_mr
MyReal operator""_mr(const char* x, std::size_t n);
    // Hvata ovo: "a"_mr, "Pera"_mr
MyReal operator""_mr(const char* x);
    // Hvata ovo: 0xDEDABABA_mr i dobija tačno taj string
    // Korisno npr. kada imamo bolju preciznost ili veći opseg od
    // long double ili long long
    // 90'223'372'036'854'775'808_mr
```

Korisnički literali

- Obično nemamo razloga da ne koristimo constexpr.

```
class MyReal {
  ...
};

constexpr MyReal operator"" _mr(long double x);

void main() {
  std::cout<< 9.0_mr;
}
```

RT-RK
COMPUTER BASED SYSTEMS

Move semantika

Metode koje se automatski generišu -> C++98

- Konstruktor (poziva se pri stvaranju promenljive)
- Konstruktor kopije (poziva se, između ostalog, pri prosleđivanju parametara funkcije i vraćanju povratne vrednosti)
- Dodela kopije (predstavlja dodelu vrednosti jednog objekta drugom objektu istog tipa)
- Destruktor (kada promenljiva završi svoj životni vek)
- Dva su razloga zašto se ove funkcije automatski generišu:
  - Te operacije su toliko česte da vrlo retko neka od njih ne treba.
  - Da bi ponašanje struktura koje sadrže samo podatke članove (atribute) ostalo isto kao u C jeziku.

Metode koje se automatski generišu -> C++98

- To znači da su ove dve definicije podjednake:

```
struct Token {
 char kind;
 double value;
};
```

```
struct Token {
  Token() {}
  Token(const Token& x)
    : kind(x.kind), value(x.value) {}
  Token& operator=(const Token& x) {
    kind = x.kind; value = x.value;
  }
  ~Token() {}
  char kind;
  double value;
};
```

Metode koje se automatski generišu -> C++98

- Posebno su interesantne ove tri metode:
  - **Konstruktor kopije** (poziva se, između ostalog, pri prosleđivanju parametara funkcije i vraćanju povratne vrednosti)
  - **Dodela kopije** (predstavlja dodelu vrednosti jednog objekta drugom objektu istog tipa)
  - **Destruktor** (kada promenljiva završi svoj životni vek)

- Pravilo trojke: „Ako vam ne odgovara podrazumevana verzija bar jedne od ove tri metode, onda vam najverovatnije ne odgovara podrazumevana verzija ni jedne od njih.“
- To jest: „Najčešće ćeš definisati ili sve tri, ili nijednu.“

Metode koje se automatski generišu

- U određenim slučajevima ne želimo da imamo neku od ovih metoda.
- Za podrazumevani **konstruktor** je dovoljno da se definiše konstruktor koji prima neke parametre.

```
struct Token {
  Token(char ch, double val) : kind(ch), value(val) {}
  Token(char ch) : kind(ch) {}
  char kind;
  double value;
};
Token x; // GREŠKA
// Mora ovako
Token y('8', 9.5); // ili ovako Token y('8')
```

=default

- A ako ipak treba i podrazumevani prazni konstruktor, onda može i ovako:

```
struct Token {
  Token() = default;
  Token(char ch, double val) : kind(ch), value(val) {}
  Token(char ch) : kind(ch) {}
  char kind;
  double value;
};
Token x; // // sada može i ovo
```

Metode koje se automatski generišu

- U određenim slučajevima ne želimo da imamo neku od ovih funkcija.
- **Konstruktor kopije** i **dodela kopije** mogu da se deklarišu kao privatni.

```
struct Token {
   char kind;
   double value;
private:
   Token(const Token& x);              // ne treba definicija
   Token& operator=(const Token& x);// ne treba definicija
};
```

- Ali to ima nekoliko mana ...

=delete

- A sada može i ovako

```
struct Token {
  Token(const Token& x)= delete;
  Token& operator=(const Token& x) = delete;
  char kind;
  double value;
};
```

=delete, =default

- Ukratko, ove dve konstrukcije nam omogućavaju da eksplicitno navedemo kako želimo sprega naše klase da izgleda
- Na primer:

```
struct Token {
  Token() = delete;
  Token(char ch, double val) : kind(ch), value(val) {}
  Token(const Token& x) = default;
  Token& operator=(const Token& x) = delete;
  ~Token() = default;
  char kind;
  double value;
};
```

=delete - dodatna upotreba

- Ova konstrukcija ima još jednu upotrebu.
- Postoje podrazumevane konverzije osnovnih tipova, npr:

```
void foo(long x);
foo(5.0); // može i ovako da se zove
int a;
foo(a);   // isto OK
```

- Ali ako želimo da to zabranimo

```
void foo(long x);
void foo(int x) = delete;
void foo(double x) = delete;
foo(5.0); // sada ovo ne može
int a;
foo(a);   // takođe ne može
```

Delegiranje konstruktora

- Do sada konstruktori nisu mogli da pozivaju druge konstruktore.

```
class Rectangle : public Shape {
  Point x;
  int w;
  int h;
public:
  Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
      if (h <= 0 || w <= 0) error("Bad arguments");
  }
  Rectangle(Point a, Point b) : x(a), w(b.x-a.x), h(b.y-a.y) {
      if (h <= 0 || w <= 0) error("Bad arguments");
  }
};
```

Delegiranje konstruktora

- Problem se delimično mogao rešavati na sledeći način:

```
class Rectangle : public Shape {
  Point x;
  int w;
  int h;
  void check() {
    if (h <= 0 || w <= 0) error("Bad arguments");
  }
public:
  Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
    check();
  }
  Rectangle(Point a, Point b) : x(a), w(b.x-a.x), h(b.y-a.y) {
    check();
  }
};
```

Delegiranje konstruktora

- Ali, sada konstruktori mogu da zovu druge konstruktore:

```
class Rectangle : public Shape {
  Point x;
  int w;
  int h;
public:
  Rectangle(Point xy, int ww, int hh) : x(xy), w(ww), h(hh) {
    if (h <= 0 || w <= 0) error("Bad arguments");
  }
  Rectangle(Point a, Point b) : Rectangle(a, b.x-a.x, b.y-a.y) {
  }
};
```

Problem koji rešavamo

- Neka imamo korisničku klasu matrice elemenata tipa double

```
class Matrix {
public:
  Matrix() {};
  ...
  ~Matrix() { delete[] data; }
  Matrix(const Matrix& x);
  Matrix& operator=(const Matrix& x);
private:
  double* data = nullptr;
  size_t size = 0;
};
```

Prenošenje matrice

- Posmatrajmo ovu funkciju:

```
Matrix operator+(Matrix a, Matrix b) {
  Matrix res;
  // Saberi matrice i rezultat smesti u res
  return res;
}

Matrix x, y, z;
z = x + y; // Koliko puta se kopiraju matrice?
```

Prenošenje matrice

- Ulazne matrice se bespotrebno kopiraju.
- Kompajler može optimizovati kod tako što će ukolniti bespotrebna kopiaranja, ali se na to ne možemo oslanjati u opštem slučaju.
- Rešenje za ulazne parameter:

```
Matrix operator+(const Matrix& a, const Matrix& b);
```

- Šta je sa povratnom vrednošću?

Prenošenje matrice

- Jedna ideja:
  - Vraćamo pokazivač na objekat zauzet pomoću **new:**

```
Matrix* operator+(const Matrix& a, const Matrix& b);
Matrix& z = *(x + y)
```

- Problemi:
  - Ružno na mestu poziva.
  - Ko zove **delete?**

Prenošenje matrice

- Druga ideja:
  - Vraćamo referencu na objekat zauzet pomoću **new:**

```
Matrix& operator+(const Matrix& a, const Matrix& b);
Matrix& z = x + y;
```

- Problemi:
  - ~~Ružno na mestu poziva.~~
  - Ko zove **delete?**

Prenošenje matrice

- Treća ideja:
  - Prosleđujemo referencu na već zauzeti objekat u koji treba da se smesti rezultat:

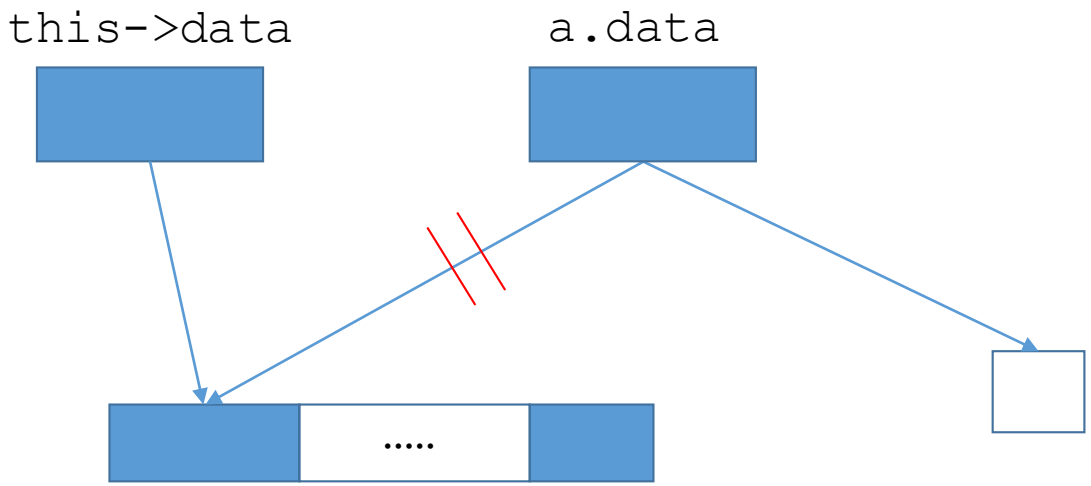
```
void operator+(const Matrix& a, const Matrix& b, Matrix& res);
Matrix res = x + y;
void plus(const Matrix& a, const Matrix& b, Matrix& res);
plus(x, y, res);
```

- Problemi:
  - Ružno na mestu poziva.
    - A i operator sabiranja prima samo dva parametra!
  - ~~Ko zove delete?~~

Move semantika

- Nova mogućnost u C++11: Move konstruktor

```
class Matrix {
  // …
  Matrix(Matrix&& a)
  {
    data = a.data;
    a.data = nullptr;
    size = a.size;
    a.size = 0;
  }
};
Matrix z = x + y;
```



Move semantika

- A može i move operator dodele

```
class Matrix {
  // …
  Matrix& operator=(Matrix&& a) {
    delete[] data;
    data = a.data;
    a.data = nullptr;
    size = a.size;
    a.size = 0;
    return *this;
  }
};
Matrix z;
z = x + y;
```

Move semantika

- Sada matrica može ovako da izgleda

```
class Matrix {
public:
  Matrix() {};
  ...
  ~Matrix() { delete[] data; }
  Matrix(const Matrix& x);
  Matrix& operator=(const Matrix& x);
  Matrix(Matrix&&x);
  Matrix& operator=(Matrix&& x);
private:
  double* data = nullptr;
  size_t size = 0;
};
```

Metode koje se automatski generišu -> od C++11

- Ove dve definicije su podjednake:

```
struct Token {
  char kind;
  double value;
};
```

```
struct Token {
  Token() {}
  Token(const Token& x) : kind(x.kind), value(x.value) {}
  Token& operator=(const Token& x) {
    kind = x.kind; value = x.value;
  }
  Token(Token&& x) : kind(x.kind), value(x.value) {}
  Token& operator=(Token&& x) {
    kind = x.kind; value = x.value;
  }
  ~Token() {}
  char kind;
  double value;
};
```

Metode koje se automatski generišu -> od C++11

- Posebno su interesantne sledećih pet:
  - Konstruktor kopije (poziva se, između ostalog, pri prosleđivanju parametara funkciji i vraćanju povratne vrednosti)
  - Dodela kopije (predstavlja dodelu vrednosti jednog objekta drugom objektu istog tipa)
  - Konstruktor premeštanja (move konstruktor)
  - Operacija premeštanja (move operator dodele)
  - Destruktor (kada promenljiva završi svoj životni vek)

- Pravilo **petice:** „Ako vam ne odgovar podrazumevana verzija bar jedne od ovih pet funkija, onda vam najverovatnije ne odgovara podrazumevana verzija ni jedne od njih.“
- To jest: „Najčešće ćeš definisati ili svih pet funkcija, ili ni jednu“.

Pravila pod kojim se metode automatski generišu

- Deo pravila preuzet iz C++98 (ali označen kao „deprecated“ od C++11):
- Konstruktor kopije, dodele kopije i destruktor će prevodilac automatski generisati ukoliko ih korisnik ne implementira i ukoliko su potrebne.

- Pravilo vezano za move operacije:
- Move konstruktor i move operator dodele će biti automatski generisani samo ukoliko nije deklarisana ni jedna od ovih 5 specijalnih metoda.
  - Čak iako samo deklarišemo neku od ovih metoda sa =delete ili =default, neće biti automatski generisani!
- Takođe, ako deklarišemo bar jednu move operaciju, copy operacije više neće biti automatski generisane.
- Preporuka je da se svih 5 metoda uvek eksplicitno deklariše makar sve metode bile označene kao =default.

Move semantika

- Move konstuktor i move operator dodele će biti implicitno pozvani u određenim slučajevima. Suštinski, onda kada kopajler jasno zna da „desna strana“ u toj naredbi završava svoj životni vek.
  - Povratna vrednost
    - return naredba je kraj funkcije i zna se da promenljive automatske trajnosti u lokalnom dosegu prestaju da žive.

```
Matrix foo() {
  Matrix res;
  ...
  return res; // ovde će biti pozvan move konstruktor
}
```

  - Kada je desna strana privremeni objekat

```
Matrix a, b, c;
a + b;      // rezultat funkcije + je privremeni objekat tipa Matrix
c = a + b; // biće pozvan move operator dodele da premesti
           //  privremeni objekat u promenljivu c
Matrix d = a + b; // biće pozvan move konstruktor da premesti
                  // sadržaj privremenog objekta u novu promenljivu d
```

&&

- Na && u deklaraciji move konstruktora i move operatora dodele može da se gleda kao na nešto što pravi razliku prema običnom konstruktoru i operaciji dodele
- Ali, u pitanju je, zapravo, jedan širi koncept.
- Da bi to razumeli, moramo prvo razumeti ova dva pojma:
  - lvalue (l-vrednost – leva vrednost)
  - rvalue (d-vrednost – desna vrednost)

&&

- lvalue (l-vrednost – leva vrednost)
  - Stvari od koji može da se uzme adresa (unarnom operacijom &).
  - Ne moraju imati ime (tj. nisu samo promenljive).

```
int* p;
*p; // itekako možemo uzeti adresu: &(*p), ali nema ime
```

- rvalue (d-vrednost – desna vrednost)
  - Stvari od kojih ne može da se uzme adresa.
  - Po pravilu nemaju ime.

```
9.0;   // literal je primer d-vrednosti
a + b; // rezultat funkcije + je d-vrednost
```

&&

- Referenca može da se odnosi samo na l-vrednosti.

```
int x;
int& a = x; // int& a{x}; može

int& b = 5; // ne može

int foo();
int& c = foo(); // ne može

void bar(int& a);

int x;
bar(x); // može

bar(5); // ne može

int foo();
bar(foo()); // ne može
```

&&

- Ali const referenca može da se veže i za d-vrednost.

```
int x;
int& a = x; // int& a{x}; može

const int& b = 5; // može

int foo();
const int& c = foo(); // može

void bar(const int& a);

int x;
bar(x); // može

bar(5); // može

int foo();
bar(foo()); // može
```

&&

- U novom C++ je uvedena i nova vrsta reference: rvalue referenca

```
int x;
int&& a = x; // int&& a{x}; ne može

int&& b = 5; // može

int foo();
int&& c = foo(); // može

void bar(int&& a);

int x;
bar(x); // ne može

bar(5); // može

int foo();
bar(foo()); // može
```

&&

- rvalue reference (d-reference) ima sledeće interesantne osobine:
  - Produžava životni vek privremenih objekata za koje se vezuje (ali sa nekim ograničenjima)

```
int foo();
int&& c = foo();
std::cout<< c;
```

  - Ima prednost pri vezivanju ukoliko preklapa funkciju koja prima konstantnu referencu na l-vrednost (klasična referenca, l-referenca).

```
void foo(const int& x); // l varijanta
foo(a); // zove l varijantu
foo(5); // zove l varijantu

// ali ako imamo ovo:
void foo(const int& x); // l varijanta
void foo(int&& x);      // d varijanta
foo(a); // zove l varijantu
foo(5); // zove d varijantu
```

std::move

- Međutim, možemo naterati poziv funkcije koja prima d-referencu za parametar koji je l-vrednost.

```
void foo(const int& x); // l varijanta
void foo(int&& x);      // d varijanta
foo(a); // zove l varijantu
foo(5); // zove d varijantu
foo(std::move(a)); // zove d varijantu
// std::move je suštinski static_cast<T&&>(a)
```

- Očekuje se da nakon ovakvog poziva promenljiva **a** bude u stanju koje omogućava njeno uništenje ili dodelu nove vrednosti.
- To sa druge strane znači da dalje korišćenje promenljive **a** treba da obuhvata samo te operacije. U suprotnom, ulazimo u nedefinisano stanje.

std::move

- Jedan primer upotrebe std::move je kod idioma swap, za bilo koji tip T.

```
// C++98
template <typename T>
void swap(T& a, T& b) {
  T tmp(a);
  a = b;
  b = tmp;
}

// Od C++11
template <typename T>
void swap(T& a, T& b) {
  T tmp(std::move(a));
  a = std::move(b);     // nova vrednost u a
  b = std::move(tmp);   // nova vrednost u b
}                       // uništenje tmp
```

&&

- && se može naći i na kraju deklaracije metode:

```
class MyClass{
   ...
   Matrix getData() &&;
   Matrix getData() &;
   ...
}
```

- U ovom slučaju se odnosi na this pokazivač (na isti način kao i const).

```
MyClass foo();
foo().getData(); // Prva metoda (&&) će biti pozvana

MyClass var;
var.getData();   // Druga metoda (&) će biti pozvana.
```

Prosleđivanje parametara

- Postoji još jedan problem u čijem rešavanju učestvuju d-reference.
- Zamislimo tip T koji ima i move konstruktor i move operator dodele.

```
void foo(T a) {
  ...
  T tmp{a};
  ...
}
T x;
foo(x);      // kopiranje
T baz();
foo(baz()); // kopiranje
```

```
void foo(const T& a) {
  ...
  T tmp{a};
  ...
}
T x;
foo(x);  // nema dodatnog kopiranja
T baz();
foo(baz()); // ali ni move konstruktora
```

- Desna strana: nema dodatnog kopiranja u prvom slučaju, ali se neće pozvati move konstruktor u drugom slučaju, iako on postoji za tip T. Na mestu stvaranja promenljive tmp vidi se samo const T& i ne razlikuje se prvi od drugog slučaja.

Prosleđivanje parametara

- Možemo napraviti dve verzije funkcije, jedna koja se poziva za d-vrednost, a druga koja se poziva za l-vrednost

```
void foo(const T& a) { // l verzija
  ...
  T tmp{a};
  ...
}
void foo(T&& a) {      // d verzija
  ...
  T tmp{std::move(a)}; // mora ovako, jer bi sad a trajalo do
  ...                  // kraja funkcije
}
T x;
foo(x);      // nema kopiranja
T baz();
foo(baz()); // poziva se move konstuktor
```

Prosleđivanje parametara

- Ali šta ako imamo više parametara?

```
void foo(const T& a, const T& b) {
  ... T tmp1{a}; T tmp2{b}; ...
}
void foo(const T& a, T&& b) {
  ... T tmp1{a}; T tmp2{std::move(b)}; ...
}
void foo(T&& a, const T& b) {
  ... T tmp1{std::move(a)}; T tmp2{b}; ...
}
void foo(T&& a, T&& b) {
  ... T tmp1{std::move(a)}; T tmp2{std::move(b)}; ...
}
```

Prosleđivanje parametara

- U pomoć dolaze šabloni i pravila za zaključivanje tipova referenci.

```
template<typename T>
void foo(T&& a, T&& b) {
   ...
   T tmp1{std::forward<T>(a)};
   T tmp2{std::forward<T>(b)};
   ...
}
```

- Kompajler će na osnovu ovoga generisati odgovarajuću funkciju foo, za svaku kombinaciju parametara (l-vrednost ili d-vrednost)
- Ovo se zove „Savršeno prosleđivanje parametara“.
- **Kada && koristimo u kontekstu gde se zaključuju tipovi (auto ili šabloni) onda to nazivamo „prosleđivačke reference“ (ili „univerzalne reference“).**
  - U svim ostalim kontekstima, u pitanju su desne reference!

Prosleđivanje parametara

- Ovo je često kod konstruktora i fabričkih funkcija

```
class MyType
{
public:
  template<typename T1, typename T2>
  MyType(T1&& a, T2&& b)
    : m_x(std::forward<T1>(a), m_y(std::forward<T2>(b) {}
private:
  SomeType1 m_x;
  SomeType2 m_y;
};
```

RT-RK
COMPUTER BASED SYSTEMS

Pametni pokazivači

Pokazivači i upravljanje memorijom

- Jedan od najčešćih problema kod upotrebe „sirovih pokazivača“ za upravljenje memorijom je curenje memorije.

```
void foo() {
  int* niz = new int[100];
  ...
  // Izuzetak na ovom mestu ili neočekivani return
  // bi doveli do curenja memorije.
  ...
  delete [] niz;
}
```

- Ipak, pre C++11 standard nije obezbeđivao pravu alternativu.

Pametni pokazivači kao rešenje

- Šta su to pametni pokazivači?
- Objekti koji su inicijalizovani sa resursom (RAII pristup), sintaksno se ponašaju kao pokazivači, a oslobađaju resurs u svom destruktoru.
- Neuspeo pokušaj u C++98 std::auto_ptr (deprecated).

```
{
  std::auto_ptr<int> tmp(new int);
  *tmp= 5;
    ...
} // Pri izlasku iz opsega, lokalna promenljiva tmp se
  // uništava, a ona u svom destruktoru poziva delete
```

- Nedostatak move semantike u to vreme je izrodilo neobično ponašanje - kopiranje std::auto_ptr je vršio premeštanje resursa.
- Nije bilo moguće imati std::vector<std::auto_ptr<int>>

std::unique_ptr

- Naslednik auto_ptr klase.
- std::uniqe_ptr predstavlja ekskluzivno vlasništvo nad resursom.
- Moguće ga je premeštati, ali ne i kopirati.

```
{
  std::unique_ptr<MojTip> p {new MojTip(1, 2)};
  // p je sada „vlasnik“ objekta
  std::cout << p->x;
  bar(*p);
  p++;  // greška!
  p[5]; // greška!
  auto q = p;// greška, kopiranje nije dozvoljeno!
  auto r = std::move(p); // premeštanje je u redu, p
                         // više nije vlasnik resursa!
}
```

std::unique_ptr

- Veličina klase std::unique_ptr je (najčešće) ista kao i veličina običnog pokazivača.
  - Ako definišemo proizvoljnu delete funkciju koja će se koristiti u destruktoru, klasa će sadržati još jedan pokazivač na tu funkciju. Tip delete funkcije u tom slučaju postaje deo tipa uniqe_ptr-a.
- Od C++14 imamo na raspolaganju std::make_uniqe za pravljenje novih objekata:

```
std::unique_ptr<MojTip> p = std::make_unique<MojTip>(1, 2);
// ili
auto q = std::make_unique<MojTip>(3, 4);
```

- Omogućava nam da skoro potpuno izbacimo new i delete iz upotrebe!

std::unique_ptr

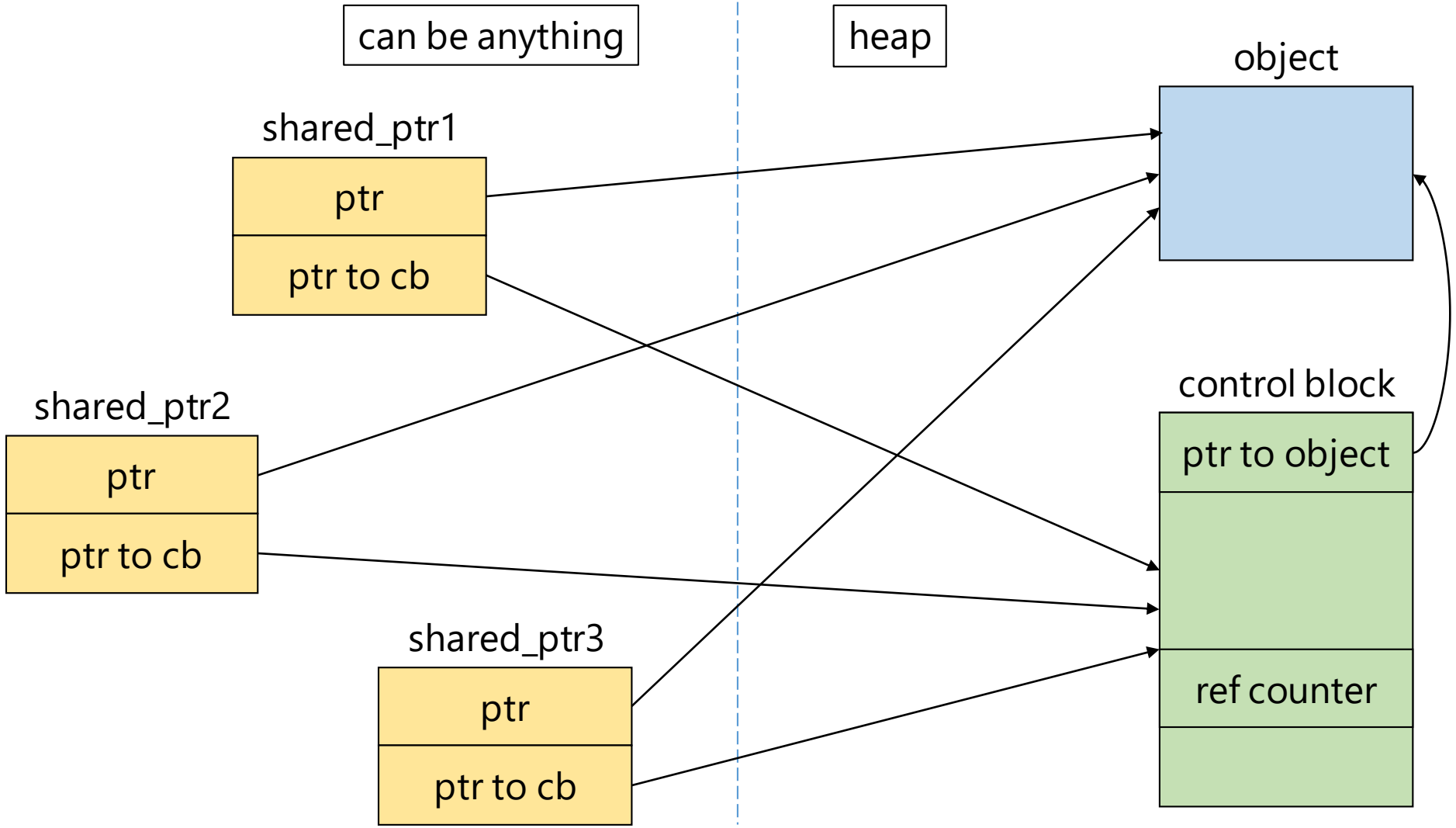
- Klasa std::unique_ptr može da se koristi i za rad sa nizovima.

```
std::unique_ptr<int[]> pniz(new int[1000]);// ispravno!
std::unique_ptr<int> qniz(new int[1000]); // ne ovako!

pniz[5] = 10; // Sada je OK, preklopljen je operator[]
              // kada su nizovi u pitanju.
```

std::shared_ptr

- Nekada je potrebno da postoji više vlasnika istog resursa.
- Ko je zadužen za njegovo brisanje?
- std::shared_ptr rešava ovaj problem.
- Tehnika brojanja referenci.
- Resurs se uništava tek kad poslednji vlasnik prestane da pokazuje na resurs.
- Cena ovoga je da svaki std::shared_ptr u sebi sadrži pokazivač na resurs i pokazivač na kontrolni blok. Takođe, i sam kontrolni blok zauzima prostor u memoriji.
- Kontrolni blok sadrži brojač referenci i pokazivač na resurs.
- Povećanje i smanjenje broja referenci mora biti atomično.
- Dobro radi uz std::uniqe_ptr.
- Brisajuća funkcija nije deo tipa objekta već kontrolnog bloka te ona ne utiče na veličinu shared_ptr-a. To je značajno prilikom grupisanja pametnih pokazivača u kontenjere.

std::shared_ptr

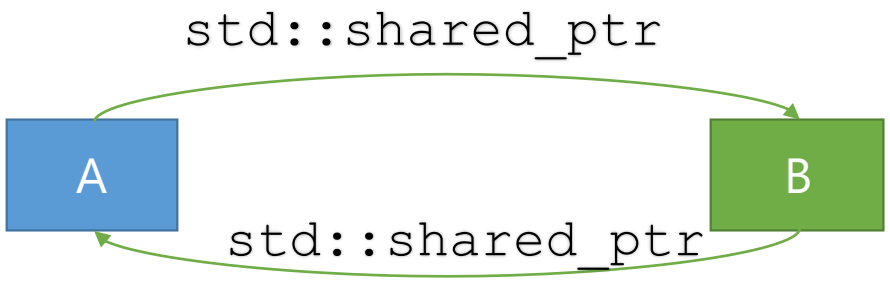


std::shared_ptr

```
void foo() {
  std::shared_ptr<MojTip> p = std::make_shared<MojTip>(1, 2);
  // broj referenci: 1 - p.use_count()

  std::shared_ptr<MojTip> q{p};  // broj referenci : 2

  {
    auto r = p;                  // broj referenci : 3
  }                              // broj referenci : 2

  p = nullptr;                   // broj referenci : 1
}                                // broj referenci : 0 - zove
                                 // se delete
```

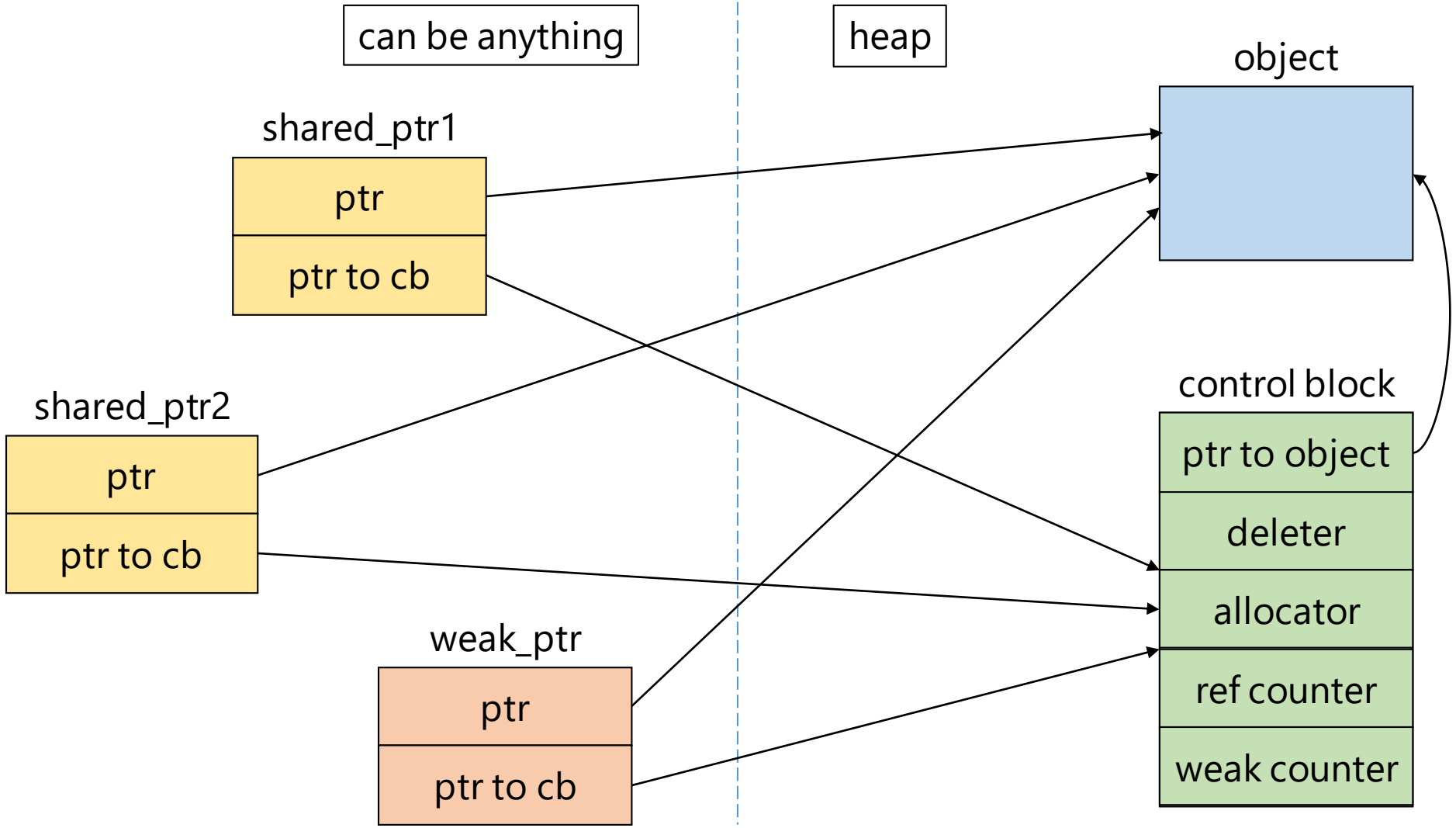
std::weak_ptr

- Jedan od problema koji imamo kod brojanja referenci – kružno referenciranje.



- Napravljen je „slabi pokazivač“ koji pomaže da se razreši taj krug.
- std::weak_ptr nije vlasnik objekta, ali može privremeno da postane.
- Detektuje da li resurs još uvek postoji.
- Dodatni brojač referenci u kontrolnom bloku.
- Resurs se oslobađa kada je brojač referenci jednak nuli.
- Sa druge strane, sam kontrolni blok se oslobađa tek kada su i brojač referenci i weak brojač referenci jednaki nuli.

std::weak_ptr



std::weak_ptr

```
std::shared_ptr<MojTip> makeMojTip();
void foo() {
  std::weak_ptr<MojTip> p = makeMojTip();
  // p nije vlasnik objekta, ali pokazuje na njega
  p.lock()->do_something(); // p će biti vlasnik tokom
                            // izvršavanja ove linije
}
weak_ptr<MojTip> q = nullptr; // ovo ne može
if (q == nullptr) ...         // ni ovo!
if (q.expired()) ...          // mora ovako!
```

Korišćenje

- Da li upotreba pametnih pokazivača eliminiše potrebu za „sirovim“ pokazivačima? Odgovor je: NE!
- Pametne pokazivače koristiti kada ste vlasnik resursa.
- Kada je potrebno proslediti pokazivač kao atribut funkciji, najčešće ćete proslediti „sirovi“ pokazivač koji pokazuje na isti objekat (možete ga dobiti std::unique_ptr::get()).
  - std::unique_ptr možete proslediti samo ako ga premestite.
  - std::shared_ptr možete prosleđivati, ali obratite pažnju na projač referenci kada se kreira lokalna promenljiva. Može i referenca.
- Koji tip pametnog pokazivača koristiti?
- Obično se kaže da bi prvo trebalo imati u vidu std::unique_ptr.

Korišćenje

- Za pravljenje pametnih pokazivača preporučuju se fabričke funkcije standardne biblioteke std::make_uniqe<T> (od C++14) i std::make_shared<T>.

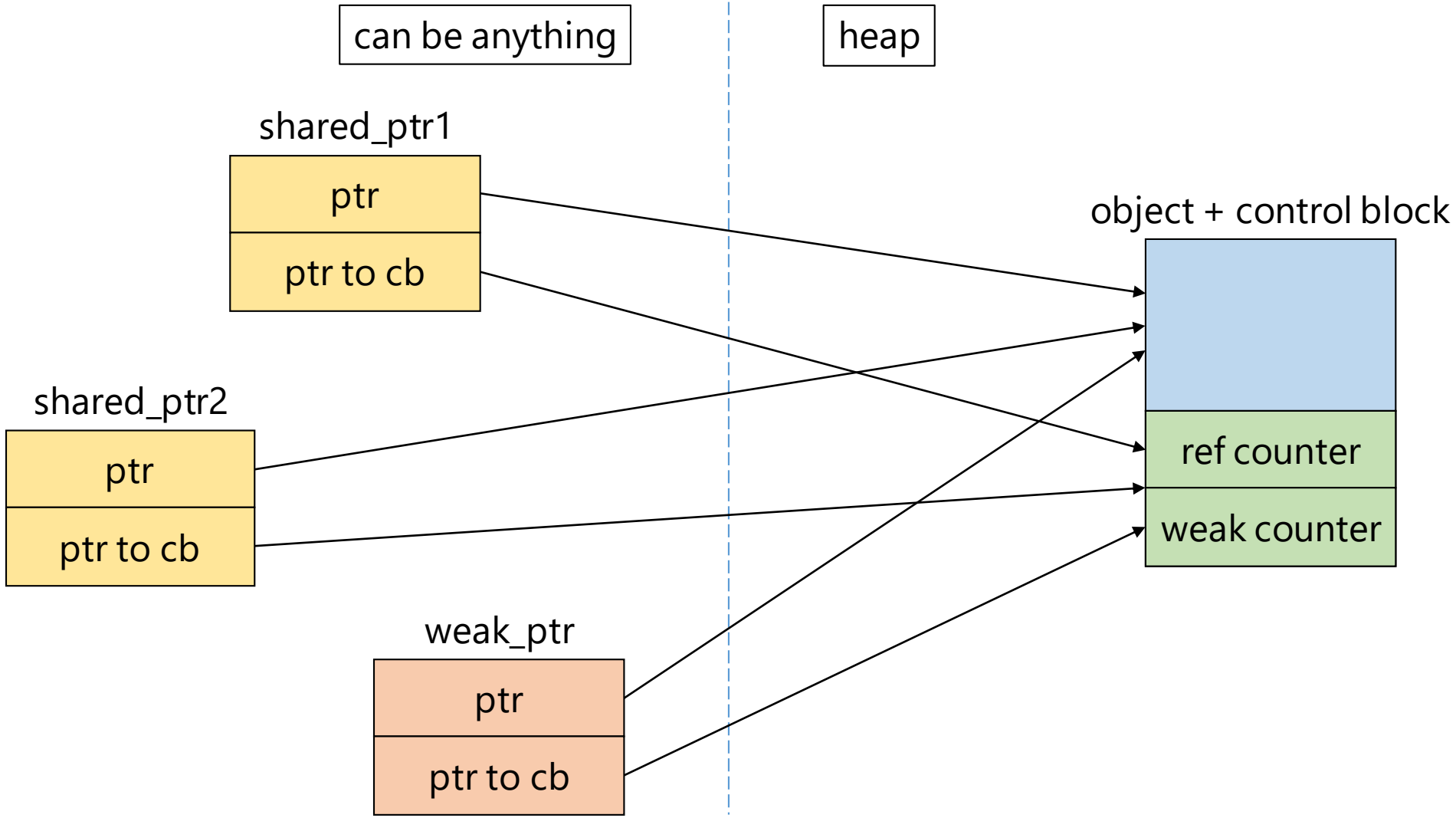
```
auto ups1(std::make_uniqe<Shape>());
std::unique_ptr<Shape> ups2(new Shape);
auto sps1(std::make_shared<Shape>());
std::shared_ptr<Shape> sps2(new Shape);
```

- Otporne su na potencijalno curenje memorije:

```
process(std::shared_ptr<Shape>(new Shape), // Potencijalno
        computeSpace());                   // curenje memorije!
```

- std::make_shared alocira memoriju i za objekat i za kontrolni blok (jedna alokacija umesto dve što smanjuje statičku veličinu programa, a samim tim i kod je efikasniji za tu jednu alokaciju).

std::make_shared



Korišćenje std::shared_ptr

- Kada koristimo make_shared i kada pravimo shared_ptr na osnovu pokazivača ili unique_ptr-a uvek alociramo kontrolni blok.

```
auto ps = new Shape;
std::shared_ptr sps1(ps); // napravi kontrolni blok za *ps.
std::shared_ptr sps2(ps); // napravi drugi kontrolni blok za *ps.
```

- Kako bi koristili isti kontrolni blok, tj. kako bi samo uvećali brojač referenci u kontrolnom bloku, konstruktoru novog shared_ptr-a potrebno je prosledi postojeći shared_ptr.

```
auto ps = new Shape;
std::shared_ptr sps1(ps);    // napravi kontrolni blok za *ps.
std::shared_ptr sps2(sps1); // sps2 koristi isti kontrolni
                            // blok kao sps1.
```

Korišćenje std::shared_ptr

- U nekim situacijama je potrebno napraviti **shared_ptr** od pokazivača **this**.

```
std::vector<std::shared_ptr<Shape>> processedShapes;
class Shape {
public:
   ...
   void process();
   ...
};
void Shape::process() {
   precessedShapes.emplace_back(this);
}
```

- Ovakva implementacija iziskuje suvišnu alokaciju dodatnih kontrolnih blokova.

Korišćenje std::shared_ptr

- API za upravljanje shared_ptr ima proširenje baš za ovakvu situaciju.

```
class Shape: public std::enable_shared_from_this<Shape> {
public:
   ...
   void process();
   ...
};
void Shape::process() {
   ...
   precessedShapes.emplace_back(shared_from_this());
}
```

- Ime ovog uzorka za projektovanje koji se ovom prilikom koristi je *The Curiously Recurring Template* pattern (CRTP).

Korišćenje

- Nedostatak je to što ne postoji podrška za pravljenje objekta sa posebnom brisajućom funkcijom.
- make_shared ne rade baš najbolje sa objektima koji imaju svoje new i delete metode.
- Blok memorije koji je napravljen sa make_shared može se osloboditi tek kad weak referenca i broj referišućih objekta budu jednaki 0.

RT-RK
COMPUTER BASED SYSTEMS

Lambda funkcije

Funkcijski objekti

- „Funkcijski objekti“ (često i „funktori“) su promenljive („objekti“) koje se sintaksno ponašaju kao funkcije, odnosno mogu tako da se upotrebljavaju.
- To se obezbeđuje preklapanje operatora ().
- Preklapanje operatora () je slično preklapanju operatora [], osim što kod operatora () možemo imati više parametara.

```
class FunctionObjectShowroom{
public:
  int operator()(int x);
  void operator()(int x, double y);
  int* operator()(float x, int y, long z);
};
FunctionObjectShowroom foo;
auto a = foo(6);
foo(a, 10.5);
auto c = foo(7.5f, a, 46L);
```

Funkcijski objekti

- Tip funkcije čini informacija o broju i tipu parametara i tipu povratne vrednosti. Dakle, dve funkcije koje primaju jednak broj parametara jednako tipa, i vraćaju povratnu vrednost jednakog tipa, jesu funkcije koje se ne razlikuju po tipu.

```
int foo(double x, long y);
int bar(double a, long b);

int(*p)(double, long) = foo;
p(5.5, 7L); // poziva se foo(5.5, 7L);
p = bar;
p(5.5, 7L); // poziva se bar(5.5, 7L);
```

- bar i foo su istog tipa!

Funkcijski objekti

- Sa druge strane, dva funkcijska objekta mogu preklapati podjednak oblik operatora () („primaju jednak broj parametara jednakog tipa, i vraćaju povratnu vrednost jednakog tipa“), ali ne moraju biti istog tipa.

```
class A {
  int operator()(double x, long y);
};
class B {
  int operator()(double x, long y);
};
A x, y;
x(5.5, 7L); // poziva se A::operator()(5.5, 7L);
B z;
z(5.5, 7L); // poziva se B::operator()(5.5, 7L);
```

- Funkcijski objekti x i y su istog tipa (A), ali z je različitog tipa (B)!

Funkcijski objekti

- Funktori i klasične funkcije ne mogu se prosleđivati istim funkcijama, jer su različitog tipa.

```
void foo(int x);
void bar(int x);
void apply(void (*f)(int)) {
  ... f(elem); ...
}
apply(foo); // OK
apply(bar); // OK
apply(foA); // ne može!
```

```
class A {
  void operator()(int x);
};
class B {
  void operator()(int x);
};
A foA; B foB;
void apply(A f) {
  ... f(elem); ...
}
apply(foA); // OK
apply(foB); // ne može!
```

Funkcijski objekti

- Ali funktori i klasične funkcije mogu instacirati iste šablone, jer se isto sintaksno ponašaju.

```
void foo(int x);
void bar(int x);
A foA;
B foB;


template<typename TFO>
void apply(TFO f) {
   ...
   f(elem);
   ...
}
```

```
class A {
   void operator()(int x);
};
class B{
   void operator()(int x)
};
```

Funkcijski objekti

```
template<typename TFO>
void apply(TFO f) {
   ...
   f(elem);
   ...
}

apply(foo); // jedna instanca, TFO je void(*)(int)
apply(bar); // ista instanca kao u prethodnom redu
apply(foA); // druga instanca, TFO je class A
apply(foB); // treća instanca, TFO je class B
```

- Zbog ovoga instaciranje šablona funktorima može dovesti do bržeg koda, jer kompajer tačno zna koja funkcija će se primeniti. Sa običnim funkcijama ne zna se da li je to foo ili bar ili nešto treće.

Funkcijski objekti

- Još jedna pogodnost funkcijskih objekata je što se mogu lako parametrizovati, jer su suštinski obične klase i kao takve mogu imati atribute.
- Ilustrovano na primeru instaciranja šablona funkcije std::find_if iz standardne biblioteke, koja vraaća prvi element iz zadatog kontejnera koji zadovoljava neki uslov (predikat):

```
class Less_than {
  int val;
public:
  Less_than(const int& x) : val(x) {}
  bool operator()(const int& x) const {
    return x < val;
  }
};
p = std::find_if(v.begin(), v.end(), Less_than(43));
p = std::find_if(v.begin(), v.end(), Less_than(76));
```

Funkcijski objekti

- Funktori se često upotrebljavaju u radu sa algoritmima iz standardne biblioteke.

```
struct CmpByName{
  bool operator()(const Rec& a, const Rec& b) const
      { return a.name < b.name; }
};
struct CmpByAge{
  bool operator()(const Rec& a, const Rec& b) const
      { return a.age < b.age; }
};
std::vector<Rec> vr;
...
std::sort(vr.begin(), vr.end(), CmpByName()); // uredi po imenu
std::sort(vr.begin(), vr.end(), CmpByAge());  // uredi po godinama
```

```
struct Rec {
  std::string name;
  int age;
};
```

Lambda funkcije

- Lamda funkcije su forma funktorskog literala i olakšavaju izražavanje u mnogim slučajevima

```
struct Rec {
  std::string name;
  int age;
};
std::vector<Rec> vr;

std::sort(vr.begin(), vr.end(),
[](const Rec& a, const Rec& b){ return a.name < b.name; });

std::sort(vr.begin(), vr.end(),
[](const Rec& a, const Rec& b){ return a.age < b.age; });
```

- Jasno je navedeno šta su ulazni parametri lambda funkcije, ali gde se navodi tip povratne vrednosti?
- Zaključuje se na osnovu tipa izraza u return naredbi.

Lambda funkcije

- Kog tipa je lambda fukcija?

```
int foo(double x, long y);

typedef int(*FooType)(double, long);
// Od C++11 umesto typedef bolje koristiti using:
// using FooType = int(*)(double, long);
FooType p1 = foo;

class A {
  void operator()(int x);
};
A p2;

  ?  p3 = [](int a, int b){ return a < b; };
```

Lambda funkcije

- **auto** pomaže u tom slučaju.

```
int foo(double x, long y);

typedef int(*FooType)(double, long);
// Od C++11 umesto typedef bolje koristiti using:
// using FooType = int(*)(double, long);
FooType p1 = foo;

class A {
  void operator()(int x);
};
A p2;

auto p3 = [](int a, int b){ return a < b; };
```

Lambda funkcije

- Videli smo da se funktori mogu parametrizovati.
- Mogu i lambda funkcije.
- Lambda funkcija predstavlja dve stvari:
  - Opis nove, bezimene, klase (tipa) koja ima definisan operator () na odgovarajući način – zvaćemo je „lambda klasa“, ili „lambda tip“ (na engleskom to zovu „closure type/class“)
  - Instanciranje te klase – zvaćemo tu instancu „lambda objekat“ (na engleskom to zovu „closure“)
- Svaki put kada se u izvršavanju naiđe na mesto gde se koristi lambda funkcija, napraviće se nova instanca te lambda klase – lambda objekat.
- Sintaksa koju smo do sada videli definiše prostu lambda klasu, koja nema atribute, tako da će svaka instanca te klase biti uvek ista.
- U takvim slučajevima lambda objekat nikad ni neće biti stvarno fizički stvoren (jer ni nema veličinu).

Zahvatanje atributa

- Ali lambda klase mogu imati atribute. Ova dva koda imaju isto dejstvo:

```
class L {
  const int limit;
public:
  L(int x) : limit(x) {}
  void operator()(int x) { if (x < limit) std::cout << x << " "; }
};
void foo(const std::vector<int>& vec) {
  for (int i= 0; i < 20; ++i) {
    auto pred = L(i);
    for (int a : vec) pred(a);
  }
}
void foo(const std::vector<int>& vec) {
  for (inti= 0; i < 20; ++i) {
    auto pred = [i](int x) { if (x < i) std::cout << x << " "; };
    for (int a : vec) pred(a);
  }
}
```

Zahvatanje atributa

- Još jedna interesantna ilustracija:

```
for (int i = 0; i < 20; ++i) {
  auto pred= [i](int x) {
    if (x < i) std::cout << x << " ";
    i = 5; // ovo se ne neće prevesti jer se ovde odnosi na atribut lambda
           // objekta (koji se isto zove i), a taj atribut je const
  };
  for (int a : vec) pred(a);
}
for (int i = 0; i < 20; ++i) {
  auto pred = [i](int x) mutable {
    if (x < i) std::cout << x << " ";
    i = 5; // ovo će se sad prevesi, ali i dalje je u pitanju atribut
           // lambda objekta tako da promena utiče samo na pred, ne i
           // u gornjoj for petlji.
  };
  for (int a : vec) pred(a);
}
```

Zahvatanje atributa

- Videli smo tzv. zahvatanje atributa po vrednosti.
- Međutim, atribut može biti referenca na nešto.
- To se zove zahvatanje (atributa) po referenci.

```
for (int i = 0; i< 20; ++i) {
  auto pred = [&i](int x) {
    if (x < i) std::cout << x << " ";
    i = 5; // ovo i dalje označava atribut lambda objekta,
           // ali koji je sada referenca na "i" iz for petlje
           // tako da će i "i" u for petlji biti promenjeno!
  };
  for (int a : vec) pred(a);
}
```

Zahvatanje atributa

- Zahvatati se mogu samo promenljive automatske trajnosti (lokalne promenljive koje nisu deklarisane ključnom reči static).
- Promenljive statičke trajnosti (globalne promenljive i lokalne sa ključnom reči static) se ne mogu zahvatiti.
- Ali to je zato što nema porebe - njima pristupamo bez zahvatanja, kao i iz bilo koje druge metode.

```
int global;
void foo(const std::vector<int>& vec) {
  for (int i= 0; i < 20; ++i) {
    auto pred = [i](int x) {
      if (x < i) std::cout << x << " " << global;
    };
    for (int a : vec) pred(a);
  }
}
```

Zahvatanje atributa

- Ostaje pitanje zahvatanja atributa objekata neke klase, kada se lambda definiše u kontekstu te klase, tj. unutar metode te klase.
- Ako je u pitanju static atribut, onda je to isto promenljiva statičke trajnosti pa je odgovor isti kao i na prethodnom slajdu.
- Ostali atributi se ne mogu zahvatiti... bar ne direktno i eksplicitno.

```
class Test {
  int y;
public:
  int x;
  void bar() {
    // auto ttt = [x, y](){ std::cout << x << y; };
    auto ttt = [this](){ std::cout << x << y; };
    ttt();
  }
};
```

Zahvatanje atributa

- Ali, kada zahvatimo this, njega jesmo zahvatili po vrednosti (i samo tako i možemo da ga zahvatimo - **&this** je sintaksna greška), ali kroz njega direktno pristupamo atributima spoljne klase, pa ih možemo i menjati.
- Slično kao što iz metode klase ne moramo stalno pisati this->attr, da bi pristupili atributu attr.

```
class Test {
  int y;
public:
  int x;
  void bar() {
    auto ttt = [this](){ x = 5; std::cout << x; };
    ttt(); // ispisaće 5, a to će biti vrednost x-a na dalje
  }
};
```

Zahvatanje atributa

- Od C++17 je moguće zahvatiti *this.
- Na ovaj način zahvatamo (const) kopiju trenutnog objekta.

```
class Test {
  int x;
  void bar() {
    auto ttt= [*this]() {
      // x = 5; // ovo ne može
      std::cout << x; };
    ttt();
  }
  void foo() {
    auto ttt = [*this]() mutable {
      x = 5;// ovo sada može, ali neće promeniti this->x
      std::cout << x; };
    ttt();
  }
};
```

Zahvatanje atributa

- Dodatna olakšica: moguće je navesti podrazumevani način zahvatanja.

```
class Test {
  int m_x, m_y;
  void bar() {
    int a, b;
    auto L1 = [a, this]() { ... std::cout << a << m_x; ... };
    // je isto kao da smo napisali:
    auto L1 = [=](){ ... std::cout << a << m_x; ... };

    auto L2 = [&a, this]() { ... a = 5; m_x = 6; ... };
    // je isto kao da smo napisali:
    auto L2 = [&](){ ... a = 5; m_x = 6; ... };

    // može i ovako: a po referenci, ostalo po vrednosti
    auto L3 = [=, &a](){ ... std::cout << b; ... };
    // ili obrnuto: a po vrednosti, ostalo po referenci
    auto L4 = [&, a](){ ... std::cout << b; ... };
  }
};
```

ovde sada nema =

Zahvatanje atributa

- Deklarisanje podrazumevanog načina zahvatanja je obično vrlo opasno, jer može dovesti do neželjenog zahvatanja.
- Zato, **vrlo pažljivo sa tim!**
- Obično je mnogo bolje eksplicitno naveti šta zahvatamo i kako.

Zahvatanje atributa

- U C++14 uveden je još jedan način zahvatanja atributa: **init capture**.
- Omogućava nam da deklarišemo lokalne promenljive koje inicijalizujemo zahvatanjem.

```
void foo() {
  auto printCounterAndIncrement = [cnt = 0](){ std::cout << cnt++; };
  auto x = std::make_unique<int>(5);
  [y = std::move(x)] // y je tipa std::unique_ptr<int>, novi vlasnik
  {
    std::cout<< *y << std::endl;
  }(); // Nije greškom zamenjen redosled {} i ()!
       // Kada lambda funkcija nema argumente () se mogu izostaviti.
       // () na kraju naredbe predstavljaju poziv (lambda) funkcije
       // tako da će kod biti odmah izvršen.
}
```

- Tip lokalne promenljive se određuje na osnovu tipa inicijalizatorskog izraza.

Generičke lambde

- Kao što i obične funkcije i klase mogu da budu šabloni (generičke), tako i lambda funkcije mogu biti šabloni.

```
auto addOp = [](int x, int y) { return x + y; }
addOp(5, 6); // OK
addOp("djura"s, "pera"s); // ne može

auto addOpGen = [](auto x, auto y) { return x + y; }
addOpGen(5, 6);
addOpGen("djura"s, "pera"s); // sada mogu obe naredbe
```

- Lambda klasa kod generičke lambde je sada zapravo šablon klase.

Zaključivanje povratne vrednosti

- Već smo imali pitanje tipa povratne vrednosti lambda funkcije.
- Odgovor je bio da je tip izraza u return naradbi automatski tip povratne vrednosti. (A ako nema return naredbe, onda je void)
- Ali, šta ako nam je telo funkcije složenije? Šta ako sadrži više return naredbi?
- Standard kaže da svi izrazi u svim return naredbama jedne lambda funkcije moraju imati izraz istog tipa. (Od C++14)

```
auto L1 = [](int a, int b) {
  if (a < b) return a;
  return b;
} // ovo je u redu
```

```
auto L2 = [](int a, int b) {
  if (a < b) return a;
  return 6.0;
} // ovo nije
```

Zaključivanje povratne vrednosti

- Određenom sintaksom možemo da eksplicitno iskažemo kog tipa treba da bude povratna vrednost.
- Ta sintaksa se naziva „prateći povratni tip“ (engl. „trailing return type“).

```
auto L2 = [](int a, int b) -> int {
if (a < b) return a;
  return 6.0;
} // sada je i ovo OK, 6.0 će se
  // implicitno konvertovati u int
```

Zaključivanje povratne vrednosti

- Zaključivanje tipa povratne vrednosti radi i sa običnim funkcijama.
- Pravila su ista:
  - Ako ima samo jedan return – tip njegovog izraza je povratna vrednost.
  - Ako ima više return naredbi – tipovi izraza moraju biti isti.

```
auto foo2(int a, int b) {
  if (a < b) return a;
  return 6.0;
} // ovo nije dobro

auto foo3(int a, int b) -> int {
  if (a < b) return a;
  return 6.0;
} // ovo je OK, mada smo mogli i ovako: int foo3(int a, int b)
```

RT-RK
COMPUTER BASED SYSTEMS

Standard Template Library

Upotreba lambdi u STL algoritmima

- Lambda funkcije omogućavaju da se STL algoritmi (nalaze se u zaglavljima <algorithm> i <numeric>) mnogo jednostavnije koriste.
- Tom tehnikom se vrlo složene obrade mogu izraziti vrlo jednostavno i u malo koda, a rezultujući kod je i dalje vrlo efikasan.
- Jedan primer smo već videli:

```
struct Rec {
   std::string name;
   int age;
};
vector<Rec> vr;
std::sort(vr.begin(), vr.end(),
   [](const Rec& a, const Rec& b){ return a.name < b.name; });
std::sort(vr.begin(), vr.end(),
   [](const Rec& a, const Rec& b){ return a.age < b.age; });
```

Upotreba lambdi u STL algoritmima

- Funkcija std::accumulate:

```
template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op) {
  for (; first != last; ++first) {
    init = op(init, *first); // "init op *first"
  }
  return init;
}
std::list<double>& ld;
double product = std::accumulate(ld.begin(), ld.end(),
  /* init */ 1.0, [](double x, double y){ return x * y; });
double sum = accumulate(ld.begin(), ld.end(),
  /* init */ 0.0, [](auto x, auto y){ return x + y; });
```

Upotreba lambdi u STL algoritmima

https://en.cppreference.com/w/cpp/algorithm/accumulate

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                    (1)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,                      (2)
              BinaryOperation op );
```

Computes the sum of the given value `init` and the elements in the range `[first, last)`. The first version uses `operator+` to sum up the elements, the second version uses the given binary function `op`, both applying `std::move` to their operands on the left hand side (since C++20).

op must not have side effects.	(until C++11)
op must not invalidate any iterators, including the end iterators, nor modify any elements of the range involved, and also *last.	(since C++11)

Parameters

first, last - the range of elements to sum

init - initial value of the sum

op - binary operation function object that will be applied. The binary operator takes the current accumulation value a (initialized to `init`) and the value of the current element b.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The type `Type1` must be such that an object of type `T` can be implicitly converted to `Type1`.
The type `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type2`. The type `Ret` must be such that an object of type `T` can be assigned a value of type `Ret`.

Type requirements

- `InputIt` must meet the requirements of *LegacyInputIterator*.
- `T` must meet the requirements of *CopyAssignable* and *CopyConstructible*.

Return value

1) The sum of the given value and elements in the given range.
2) The result of left fold of the given range over op

Upotreba lambdi u STL algoritmima

- Funkcija std::inner_product:

```
template<class In, class In2, class T , class BinOp , class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2) {
  while (first != last) {
    init = op(init, op2(*first, *first2));
    ++first;
    ++first2;
  }
  return init;
}
std::list<double> ld;
std::vector<double> vd;
double scalar_product = std::inner_product(ld.begin(), ld.end(), vd.begin(),
                                           /* init */ 0.0,
                                           [](auto x, auto y){ return x + y; },
                                           [](auto x, auto y){ return x * y; });
```

Upotreba lambdi u STL algoritmima

- Funkcija std::for_each.

```
template<class In, class UnOp>
UnOp for_each(In first, In last, UnOp f) {
    for (; first != last; ++first) {
        f(*first);
    }
    return f; // implicit move since C++11
}

std::vector<struct Rec> persons;
// Štampaj imena i godine svih osoba
std::for_each(persons.begin(), persons.end(),
    [](const Rec& rec)
        { std::cout << rec.name << " (" << rec.age << ")\n"; });
```

Upotreba lambdi u STL algoritmima

- Funkcija std::for_each treba da bude zamena za for petlju samo onda kada radimo neku operaciju nad svim elementima kontejnera.
- Češći je slučaj da hoćemo da uradimo nešto pod nekim uslovom.
- Npr. hoćemo da prebrojimo koliko osoba u listi je punoletno.

```
std::vector<struct Rec> persons;
size_t cnt = 0;
std::for_each(persons.begin(), persons.end(),
    [&cnt](const Rec& rec) { if (rec.age > 18) cnt++; });
// Radi, ali može bolje
```

- Bolje rešenje je koristiti specijalizovan algoritam za tu namenu:

```
size_t cnt = std::count_if(persons.begin(), persons.end(),
    [](const Rec& rec) { return rec.age > 18; });
```

Upotreba lambdi u STL algoritmima

- Kako izbaciti elemente iz vektora koji ispunjavaju neki kriterijum?
- Problemi koji nastaju kada izbacimo jedan element:
  - Svi elementi iza izbačenog elemenata premeštaju se za po jedno mesto „u levo“.
  - Svi iteratori, pokazivači i reference na elemente iza izbačenog se invalidiraju.
- Rešenje je „erase-remove“ idiom:

```
// Hoćemo da izbacimo sve maloletne osobe iz vektora
persons.erase(
    std::remove_if(persons.begin(), persons.end(),
                   [](const Rec& rec) { return rec.age <= 18; }),
    persons.end());
```

- std::remove_if funkcija uređuje vektor tako da su elementi koji se zadržavaju na početku vektora, a vraća iterator koji pokazuje na prvi sledeći element. Stvarno brisanje elemenata vršimo erase metodom.

Upotreba lambdi u STL algoritmima

- Još jedna primena „erase-remove“ idioma, brza transformacija stringova:

```
// Hoćemo da izbacimo sve razmake iz ulaznog stringa
std::string recenica{"Ne zelimo razmake u ovoj recenici."};

recenica.erase(
    std::remove_if(recenica.begin(), recenica.end(),
                   [](auto slovo) { return slovo == ' '; }),
    recenica.end());

std::cout << recenica << std::endl;
// Ispisuje: Nezelimorazmakeuovojrecenici.
```

Kontejneri standardne biblioteke

- Sadrži kolekciju određenih objekata.
- Upravljaju životnim vekom elemenata.
- Svaka vrsta kontenjera ima svoj način:
  - ubacivanja u kontejner
  - brisanja elemenata iz kontejnera
  - pristupa elementima kontejnera
  - prolaska kroz sve elemente kotejnera

Kontejneri do C++11

- Najpoznatije strukture koje su se do sad koristile:
  - Sekvencijalni:
    - std::vector
    - std::deque
    - std::list
  - Adaptatori kontejneri:
    - std::stack
    - std::queue
    - std::priority_queue
  - Asocijativni kontejneri:
    - std::set
    - std::multiset
    - std::map
    - std::multimap

Kontejneri od C++11

- Nove mogućnosti jezika kao što je move semantika i inicijalizacija pomoću vitičastih zagrada otvaraju prostor za proširenje starih kontejnera kao i za stvaranje novih.
- Novi kontejneri:
  - Sekvencijalni:
    - std::array – potpuna zamena za C nizove
    - std::forward_list – jednostruko ulančana lista (std::list je dvostruko ulančana)
  - Nesortirani (eng. „unordered“) asocijativni kontenjeri:
    - std::unordered_set
    - std::unordered_multiset
    - std::unordered_map
    - std::unordered_multimap

Kontejneri od C++11

- Kontejneri koji su postojali i pre C++11 su prošireni novim metodama, dok je većina postojećih metoda prkloplјena varijantama koje podržavaju move semantiku.
- Primer nove metode:
  - emplace(), emplace_back(), emplace_front() – šabloni sa promenljivim brojem parametara koje primaju prosleđivačke reference objekata koje se „savršeno prosleđuju“ konstruktoru koji pravi novi element kontejnera na odgovarajućem mestu. Posebno korisne kod sekvencijalnih kontejnera.
- Primer preklapanja metoda:
  - push(), push_back(), push_front() – ranije su postojale varijante koje su primale const T&, sada su metode preklopljene verzijama koje primaju T&& (Napomena: u pitanju je desna referenca posto se T ne zaključuje tokom poziva metode, već je zaključen prilikom pravljenja kontejnera).

Kontejneri standardne biblioteke

- Gotovo uvek je slučaj da više kontejnera može da zadovolji potrebe našeg programa.
- Koji odabrati?
- Ne donosite zaključke o performansama bez merenja!
- Benchmark testove pravite tako da po složenosti odgovaraju stvarnim zahtevima.
  - Može se desiti da za manji broj elemenata jedan tip kontejnera ima bolje performanse nego drugi, a da pri većem zauzeću bude obrnuto.
  - Meriti ono što vam je od interesa. Npr. odlučiti da li vam je bitnija brzina dodavanja i brisanja elemenata ili brzina pristupa. Takođe da li pristupamo elemetima pojedinačno ili uvek iteriramo kroz ceo kontejner i sl.

RT-RK
COMPUTER BASED SYSTEMS

Generičko programiranje

Šabloni funkcije (funkcijski šabloni, templejt funkcije)

- Da se podsetimo...
- Definicija šablonske funkcije:

```
template<class T>
T sumPow23(T x, T y) {
  return x * x + y * y * y;
}
```

- Instanciranje šablona funkcije (kompajler pravi funkciju za odgovarajući tip na osnovu zadatog šablona):

```
int r1 = sumPow23<int>(3, 4);
int r2 = sumPow23(3, 4); // isto kao i prethodna linija
```

- Parametri šablona funkcije se zaključuju iz tipova stvarnih parametara.
- Ovo je sada moguće i za šablone klasa: parametri šablona se mogu zaključiti na osnovu stvarnih parametara konstruktora.

Šabloni funkcije (funkcijski šabloni, templejt funkcije)

```
template<typename T>
T sumPow23(T x, T y) {
  return x * x + y * y * y;
}
```

- Na osnovu gornjeg šablona, ovaj kod će dovesti do toga da kompajler napravi funkciju kao na desnoj strani:

```
int r1 = sumPow23(3, 4);
```

```
int sumPow23(int x, int y) {
  return x * x + y * y * y;
}
```

- Sa drugim stvarnim parametrima napraviće se druga funkcija:

```
int r2 = sumPow23(3.0, .5);
```

```
double sumPow23(double x, double y) {
  return x * x + y * y * y;
}
```

Šabloni sa promenljivim brojem parametara

- Šabloni sa promenljivim brojem parametara omogućavaju da se napravi jedan šablon koji pokriva slučajeve sa proizvoljno mnogo parametara različitog tipa.

... označava da je u pitanju proizvoljan broj parametara, a Args je samo naziv za kasnije identifikovanje (i može biti bilo šta, ali je Args uobičajeno, kao što je T uobičajeno za jedan tip).

```
template<typename... Args>
void foo(Args... args) {
  bar(args...);
}
```

Ovo je tzv. „raspakivanje“ parametara. Označava da na tom mestu treba da se nađe lista svih funkcijskih parametara odvojenih zarezima.

Ova upotreba je vrlo specifična i označava da je lista fiktivnih parametara funkcije vezana za šablonski parametar Args. args je takođe proizvoljan, ali uobičajen naziv.

Šabloni sa promenljivim brojem parametara

```
template<typename... Args>
void foo(Args... args) {
  bar(args...);
}
```

- Ilustracija upotrebe ovakvog šablona

```
foo(5, 0.5, "hik");
// je isto što i ovo
foo<int, double, const char*>(5, 0.5, "hik");
// što čini da kompajler generiše ovakvu funkciju:
void foo(int p1, double p2, const char* p3) {
  bar(p1, p2, p3);
}
```

Šabloni sa promenljivim brojem parametara

```
template<typename... Args>
void foo(Args... args) {
  bar(args...);
}
```

- Ilustracija upotrebe ovakvog šablona

```
foo(5, 0.5, "hik");
// je isto što i ovo
foo<int, double, const char*>(5, 0.5, "hik");
// što čini da kompajler generiše ovakvu funkciju:
void foo(int p1, double p2, const char* p3) {
  bar(p1, p2, p3);
}
```

Šabloni sa promenljivim brojem parametara

- Lako se može odabrati kako se prenose parametri.

```
template<typename... Args>
void foo(const Args&... args) {
  bar(args...);
}
foo(5, 0.5, "hik");
void foo(const int& p1, const double& p2, const char* const& p3){
  bar(p1, p2, p3);
}
```

- Naravno, može i referenca i prosleđujuća referenca:

```
template<typename... Args> void foo(Args&... args) // ...
template<typename... Args> void foo(Args&&... args) // ...
```

- Na ovaj mehanizam se oslanjaju konstruktori std::thread i std::scope_lock, kao i funkcije std::make_shared i std::make_unique.

Šabloni sa promenljivim brojem parametara

- Važno je obratiti pažnju da zarezi koji se javljaju tokom raspakivanja nisu izrazi ((1, 2, 3) je, na primer, ispravan izraz, čija je vrednost 3), tj. parametri se mogu raspakovati samo u kontekstu poziva funkcija i slično.
- Još važnije je imati u vidu da se parametri respakuju po obrascu koji je zadaz:

```
template<typename... Args>
void foo(Args... args) {
  bar(2 * args...);
}
foo(5, 0.5);
// Instancira funkciju:
void foo(int p1, double p2) {
  bar(2 * p1, 2 * p2);
}
```

```
template<typename... Args>
void foo2(Args... args) {
  bar(baz(args)...);
}
foo2(5, 4);
// Instancira funkciju:
void foo2(int p1, int p2) {
  bar(baz(p1), baz(p2));
}
```

Šabloni sa promenljivim brojem parametara

- Ovakav oblik šablona gde je jedini parametar zapravo „paket“ parametara, vrlo je redak.

```
template<typename... Args>
void foo(Args... args) //...
```

- Mnogo češće imamo ovako nešto:

```
template<typename T, typename... Args>
void foo(T x, Args... args) {
  // imamo x kao prvi parametar i args kao ostali parametri
  // sada možemo rekurzivno ići kroz listu parametara
}
```

Šabloni sa promenljivim brojem parametara

- Jednostavan primer sume:
- Zelimo da nam funkcija vraća sumu svih parametara, npr:

```
int a = sum(1, 2, 3);     // a treba da bude 6
int b = sum(6, 7, 5, 2); // b treba da bude 20
```

- Implementacija bi mogla ovako da izgleda:

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
  return x + sum(args...);
}
```

Šabloni sa promenljivim brojem parametara

- Jednostavan primer sume:

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
  return x + sum(args...);
}
sum(3, 4, 5);
// Instanciraju se sledeće funkcije:
int sum(int x, int p1, int p2) { return x + sum(p1, p2); }
int sum(int x, int p1) { return x + sum(p1); }
int sum(int x) { return x + sum(); }
int sum() ??? // GREŠKA! Po šablonu mora biti bar jedan
              // parametar!
```

- ... ali problem je što sum() funkcija nije definisana!

Šabloni sa promenljivim brojem parametara

- Jedno rešenje može biti da definišemo tu nedostajuću funkciju sum(), ali tu bi naišli na nove probleme.
- Bolje rešenje je da uradimo specijalizaciju šablona za jedan parametar:

```
template<typename T>
T sum(T x) {
  return x;
}
template<typename T, typename... Args>
T sum(T x, Args... args) {
  return x + sum(args...);
}
sum(3, 4, 5);
int sum(int x, int p1, int p2) { return x + sum(p1, p2); }
int sum(int x, int p1) { return x + sum(p1); }
int sum(int x) { return x; }
```

Šabloni sa promenljivim brojem parametara

- Korisno je znati i za **sizeof...** operaciju.
- Ta operacija se primenjuje na paket parametara i vraća broj parametara.
- Ilustrovano na funkciji koja vraća srednju vrednost svojih parametara.

```
template<typename... Args>
auto average(Args... args) {
   return sum(args...) / sizeof...(args);
}
```

- Šta će biti tip povratne vrednosti funkcije average?
- Zapravo, i tip povratne vrednosti sum bi trebalo tako da definišemo, ako želimo da radi izpravno i sa raznovrsnim tipovima:

```
template<typename T> T sum(T x) { return x; }
template<typename T, typename... Args>
auto sum(T x, Args... args) { return x + sum(args...); }
double x = sum(5, 0.5, 7); // sad će i ovo raditi ispravno
```

Šabloni sa promenljivim brojem parametara

- Funkcije nastale od šablona takođe mogu biti sračunljive tokom prevođenja:

```
template<typename T>
constexpr T sum(T x) {
  return x;
}
template<typename T, typename... Args>
constexpr auto sum(T x, Args... args) {
  return x + sum(args...);
}
template<typename... Args>
constexpr auto average(Args... args) {
  return sum(args...) / sizeof...(args);
}
constexpr int x = average(5, 6, 7);
```

Torke (std::tuple)

- Torke bi bile neka vrsta šablona klase sa promenljivim brojem atributa.
- One nam omogućavaju da generički jedinstveno izrazimo sve ove klase:

```
struct X {
  int a1;
  double a2;
} x;
struct Y {
  double a1;
  std::string a2;
  long a3;
} y;
struct Z {
  long long a1;
  const double* a2;
  std::vector<int> a4;
} z;
```

Torke (std::tuple)

- Torke bi bile neka vrsta šablona klase sa promenljivim brojem atributa.
- One nam omogućavaju da generički jedinstveno izrazimo sve ove klase:

```
struct X {
  int a1;
  double a2;
} x;
struct Y {
  double a1;
  std::string a2;
  long a3;
} y;
struct Z {
  long long a1;
  const double* a2;
  std::vector<int> a4;
} z;
```

```
std::tuple<int, double> x;
// ili:
auto x = std::make_tuple(5, 0.5);

std::tuple<double,
           std::string,
           long> y;

std::tuple<long long,
           const double*,
           std::vector<int>> z;
```

std::tuple

- Postoji mogućnost i da se elementi torke „raspakuju“ u pojedinačne promenljive:

```
std::tuple<int, double, bool> x;
int a1;
double a2;
bool a3;
std::tie(a1, a2, a3) = x;
// a može i ovako ako nas neko od polja ne interesuje:
std::tie(std::ignore, a2, std::ignore) = x;

// za dohvatanje samo jednog polja može i ovako:
if (std::get<2>(x)) { // Indeks mora biti constexpr izraz
  std::cout << std::get<double>(x); // Dozvoljeno ako postoji
}                                   // tačno jedan element tipa
                                    // double
```

std::tuple

- Jedan od najčešćih načina korišćenja torki je za vraćanje više povratnih vrednosti iz funkcije:

```
std::tuple<int, float, int> foo() {
   ...
   return {5, x, y}; // do C++17 je moralo ovako da se piše:
                     // return std::tuple<int, float, int>{5, x, y};
}

int a1, a3;
float a2;
std::tie(a1, a2, a3) = foo();
```

std::tuple

- Ali, postoje i druge koristi od torki.
- U suštini, to je alternativni način pravljenja heterogenih skupova.
- Na primer:
- Želimo da imamo objekat koji predstavlja složenu funkciju koja prima ceo broj, vraća ceo broj, a koja je sledećeg oblika:

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

- gde su f_i bilo kakvi funkcijski objekti koji imaju definisan operator () koji prima ceo broj i vraća ceo broj.
- Svaki poziv operatora () tog složenog objekta nad celobrojnim parametrom **x**, treba da prosledi to **x** svim funkcijama f_i i da sumira rezultate.

std::tuple

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

Jedan način je da uspostavimo hijerarhiju tipova, gde bazna klasa ima virtuelnu metodu (i operator() može biti virtuelan) za sračunavanje funkcije $int \rightarrow int$.

- Zatim, da napravimo da naša klasa složene funkcije ima vektor (ili neki drugi kontejner) čiji elementi su pokazivači na baznu klasu, i u koji tako možemo dodavati pokazivače na objekte bilo koje klase izvedene iz bazne.
- Sračunavanje funkcije u složenoj klasi bi se onda svelo na prolazak kroz kontejner i sumiranje rezultata koje daje poziv virtuelne metode za parametar x.

std::tuple

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

```
class SumFunc {
  std::vector<BaseFunc*> m_fs;
public:
  int operator()(int x) {
    int sum{0};
    for (auto it : m_fs)
      sum += (*it)(x); // virtuelna metoda
    return sum;
  }
};
```

- Kod ovog pristupa moramo misliti o tome ko stvara/uništava funkcije u vektoru (tj. ko je njihov vlasnik), a svakako imamo oganičenje da se računaju samo funkcijski objekti klasa koje nasleđuju našu baznu klasu.

std::tuple

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

RT-RK
COMPUTER BASED SYSTEMS

- Korišćenjem torki možemo ponuditi drugačije rešenje koje ima nekoliko prednosti:

```
template<typename... Ts>
class SumFunc {
  std::tuple<Ts...> m_tuple;
  constexpr static std::size_t m_ts
      = std::tuple_size<std::tuple<Ts...>>::value;
  template<int I>
  int eval(int x) { // pomoćna šablonska funkcija koja prolazi
    ...             // kroz elemente torke, zove operator() i
  }                 // zbraja rezultat.
public:
  SumFunc(std::tuple<Ts...> x) : m_tuple(x) {}
  int operator()(int x) { return eval<m_ts>(x); }
};
```

std::tuple

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

- Sada stvaranje nove složene funkcije može da izgleda ovako:

```
auto f = SumFunc<Ts...>(std::make_tuple(
    [](int x){ return 5 * x; },
    [](int x){ return 6 + x; }));
std::cout << f(4); // ispisaće 5 * 4 + (6 + 4) = 30

auto g = SumFunc<Ts...>(std::make_tuple(
    [](int x){ return x / 2; },
    f));
std::cout << g(4); // ispisaće 4 / 2 + f(x) = 32
```

- Obratiti pažnju da promenljive f i g nisu istog tipa (različite instance istog šablona klase), te, na pimer, ne može da se uradi ovo:

```
f = g
```

Specijalizacija/razrešenje preklopljenih funkcija

- Zamislimo da imamo neki generički algoritam koji radi nešto, i usresredimo se na jedan njegov parametar (ostale parametre ćemo zanemariti u kodu):

```
template<typename T> void myAlg(T x);
```

- Ukoliko je taj parametar ceo broj onda imamo vrlo specijalan algoritam, koji je različit od generičkog i koji sjajno radi ako je to ceo broj.
- Želimo da imamo uniforman način pozivanja funkcije koja obavlja algoritam, ali da se u slučaju celobrojnog parametra u svari pozove ona specijalna funkcija.
- Mogli smo dodati i običnu funkciju samo za long long:

```
void myAlg(long long x); // specijalna verzija
int aInt;
long long aLongLong;
myAlg(aLongLong); // Biće pozvana specijalna verzija
myAlg(aInt);      // Neće biti pozvana specijalna verzija
```

Specijalizacija/razrešenje preklopljenih funkcija

- Zašto specijalna verzija nije pozvana u slučaju **int** parametara?
- Tri koraka u razrešavanju preklopljenih funkcija:
  1. Prvo se šabloni funkcija ignorišu. Ukoliko postoji redovna funkcija koja odgovara po imenu i tačno po tipu parametara (po potpisu), onda će se ta funkcija pozvati (objašnjava zašto se za **long long** zove specijalna verzija)
  2. Sada se gledaju šabloni funkcija. Ukoliko se neki šablon funkcije može instancirati da tačno odgovara tipovima parametara (daje tačan potpis), onda će se ta funkcija pozvati (i biće napravljena na osnovu tog šablona – ako već nije)
  3. Tek u trećem koraku se razmatraju redovne (nešablonske) funkcije koje bi se mogle pozvati uz primenu nekih implicitnih konverzija (kada bi se sa **int** parametrom mogla pozvati funkcija koja prima **long long**, ali u prethodnom koraku se pronalazi odgovarajuća instanca (specijalizacija) šablona i do trećeg koraka neće ni doći.

Svojstva tipova

- Dakle, treba nam da nekako izrazimo „za sve cele brojeve“, odnosno da naša funkcija radi na specijalan način **ako je tip parametra ceo broj**, a na generički način za sve ostale slučajeve.
- Prvo da vidimo kako da utvrdimo da li je nešto ceo broj.
- Standardna biblioteka nudi pomoć:

```
#include <type_traits>
template<typename T>
void myAlg(T x) {
  if (std::is_integral<T>::value)
    // if (std::is_integral_v<t>) <- ovako može od C++17
    // i oslanja se šablone promenljivih
    myAlgSpec(x); // specijalna verzija
  else
    // regularna generička verzija
}
```

Svojstva tipova

- Biblioteka <type_traits> nam nudi niz (oko 70) predikata koji se tiču tipova:

```
std::is_pointer<T>::value
std::is_floating_point<T>::value
std::is_const<T>::value
std::is_abstract<T>::value
std::is_polymorphic<T>::value
std::is_copy_assignable<T>::value
std::is_convertible<From, To>::value
std::is_base<Base, Derived>::value
```

- Kao i nekoliko (oko 25) transfomacija tipova:

```
std::remove_cv<T>::type // ovo je tip, isti kao T,
                        // samo bez const i volatile
std::add_pointer<T>::type
std::add_pointer_t<T> // od C++14
```

Svojstva tipova

- Sada možemo, na primer, napraviit šablon funkcije koji radi samo ako tip parametra može da se kopira i lepo prijaviti grešku ukoliko je šablon instanciran sa lošim parametrima

```
template<typename t>
void foo(T x) {
  static_assert(std::is_copy_assignable<T>::value
      && std::is_copy_constructible<T>::value,
      "foo expect copy assignable and copy
          constructible classes");
}
```

if constexpr

- Pogledajmo još jednom ovaj primer:

```
template<typename T> void myAlg(T x) {
  if (std::is_integral<T>::value)
    myAlgSpec(x);
  else
    // generička verzija
}
```

- Ovo su dva izgleda koda, u zavisnosti da li jeste ili nije ceo broj:

```
void myAlg(int x) {
  if (true)
    myAlgSpec(x);
  else
    // generička verzija
}
```

```
void myAlg(float x) {
  if (false)
    myAlgSpec(x);
  else
    // generička verzija
}
```

if constexpr

- Po sličnom principu možemo pokušati da ovaj kod sa početka predavanja:

```
template<typename T> T sum(T x) {
  return x;
}
template<typename T, typename... Args>
T sum(T x, Args... args) {
  return x + sum(args...);
}
```

- Napišemo na ovaj način:

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
  if (sizeof...(args) == 0) return x;
  else return x + sum(args...);
}
```

if constexpr

- Ali biće grešaka u prevođenju jer od šablona nastaju sledeće funkcije:

```
sum(3, 4);

int sum(int x, int p1) {
  if (1 == 0) return x;
  else return x + sum(p1);
}
int sum(int x) {
  if (0 == 0) return x;
  else return x + sum(); // GREŠKA! sum() nije definisano
}
```

- Obratiti pažnju da se **sum()** nikad ne bi ni pozvalo, ali tokom prevođenja mora biti definisano (ili bar deklarisano).

if constexpr

- Za ovakve slučajeve, kada imamo uslov koji je sračunljiv tokom prevođenja, imamo na raspolaganju naredbu **if constexpr:**

```
template<typename T, typename... Args>
T sum(T x, Args... args) {
  if constexpr(sizeof...(args) == 0) return x;
  else return x + sum(args...);
}
```

- Sada će funkcije nastale od šablona sum izgledati ovako:

```
sum(3, 4);
int sum(int x, int p1) {
  return x + sum(p1);
}
int sum(int x) {
  return x;
}
```

if constexpr

- I ovo je sad bolje da pišemo ovako:

```
template<typename T> void myAlg(T x) {
  if constexpr(std::is_integral<T>::value)
    myAlgSpec(x); // specijalna verzija
  else
    // regularna generička verzija
}
```

- Obratite pažnju da valjanost odbačene grane nije bitna samo u slučaju da uslov zavisi od šablonskih parametara.
- Na primer, ovo je i dalje greška u prevođenju:

```
if constexpr(false) {
  int x = “43”;
}
```

std::tuple

$$F(x) = \sum_{i=0}^{n-1} f_i(x)$$

- Sada možemo elegantno implementirati i eval metodu:

```
template<typename... Ts>
class SumFunc {
  std::tuple<Ts...> m_tuple;
  constexpr static std::size_t m_ts =
      std::tuple_size<std::tuple<Ts...>>::value;
  template<int I>
  int eval([[maybe_unused]] int x) {
    if constexpr (I == 0) return 0; // Moglo je da se ide do I = 1,
    else {                          // ova varijanta je za ilustraciju
      auto p = std::get<m_ts - I>(m_tuple); // atributa maybe_unused
      return p(x) + eval<I - 1>(x);
    }
  }
public:
  SumFunc(std::tuple<Ts...> x) : m_tuple(x) {}
  int operator()(int x) { return eval<m_ts>(x); }
};
```

RT-RK
COMPUTER BASED SYSTEMS

Višenitno programiranje

Nit?

- Nit je programski tok koji se u vremenu prepliće sa izvršavanjem drugih programskih tokova, tj. drugih niti.
- Sve niti konkurišu za zauzeće jednog procesora, pa se programi koji koristi nit zove „konkurentan program“, a pisanje takvih programa „konkurentno programiranje“.
- Paralelizam je vrlo sličan konkurenciji, samo što ima više procesora. Zato je važna tehnika.
- U tzv. managed okruženju, main funkija se izvršava u jednoj, prvoj niti.
- U većini programa to je i jedina nit.
- Ali, iz te prve niti mogu se stvoriti druge niti.

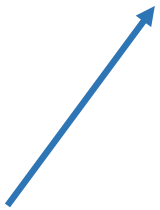
Stvaranje niti

- Od 2011. godine, niti su sastavni deo C++ sandarda, a stvaranje i upravljanje nitima se ostaruje kroz standardnu biblioteku. Biblioteka je urađena po uzoru na POSIX niti, samo u OOP maniru.
- Stvaranje niti se obavlja stvaranjem objekta tipa std::thread.
- Tada se dešava i njeno pokretanje.

```
void print1() {
  std::cout << "1";
}

void main() {
  std::thread thread1(print1);
  ...
}
```

Stvaranje niti

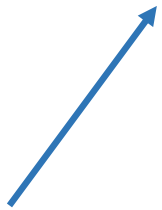
```
std::thread thread1(?????);
```



- Šta može biti parametar?

Stvaranje niti

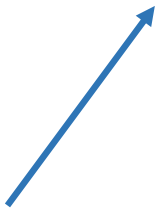
```
void print1() {
  std::cout << "1";
}

std::thread thread1(print1);
```



- Šta može biti parametar?
  - Pokazivač na funkciju

Stvaranje niti

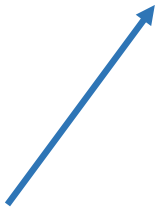
```
struct print1Type {
  void operator()() {
    std::cout << “1”;
  }
}
print1Type print1;
std::thread thread1(print1);
```



- Šta može biti parametar?
  - Pokazivač na funkciju
  - Funkcijski objekat

Stvaranje niti

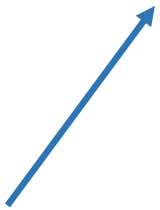
```
struct print1 {
  void operator()() {
    std::cout << “1”;
  }
}

std::thread thread1(print1{});
```



- Šta može biti parametar?
  - Pokazivač na funkciju
  - Funkcijski objekat

Stvaranje niti

```
std::thread thread1([](){ std::cout << "1"; });
```



- Šta može biti parametar?
  - Pokazivač na funkciju
  - Funkcijski objekat
  - Lambda

Prenošenje parametara funkciji niti

- Funkcija može primati parametre.

```
struct print {
  void operator()(std::string s) {
    std::cout << s;
  }
};
std::thread thread1(print{}, “1”);
std::thread thread2(print{}, “2”);
```

- Bez obzira koliko je parametara i kog su tipa.

```
struct print1 {
  void operator()(std::string s, int i) {
    std::cout << s << i;
  }
};
std::thread thread3(print1{}, “1”, 2);
```

Stvaranje niti

- Obratiti pažnju da će objekat niti (objekat koji predstavlja nit – std::thread objekat) biti stvoren odmah nakon završetka konstruktora, kao i bilo koja druga promenljiva.
- Sama nit će u tom trenutku biti napravljena i u stanju pripravnosti, ali nema garancije kad će biti pokrenuta.

```
void main() {
   std::thread thread1([]() {std::cout << "1"; });
   std::cout << "2";
   ...
}
```

- Ispis može biti 12 i 21, zavisi od toga kada će se nit pokrenuti.

Uništavanje niti

- Pogledajmo sada ovaj kod:

```
void main() {
  std::thread thread1(print1);
} // thread1 se uništava (poziva se destruktor)
```

- Šta se dešava ako nit thread1 nije završila (ili čak nije ni počela sa radom)?
- Smatra se neregularnim uništavanje objekta niti pre nego što je nit završila sa radom.
- Zbog toga je u ovom slučaju potrebno da glavna nit na neki način sačeka na završetak izvršavanja novonastale niti, pa tek onda da i ona sama završi sa radom.

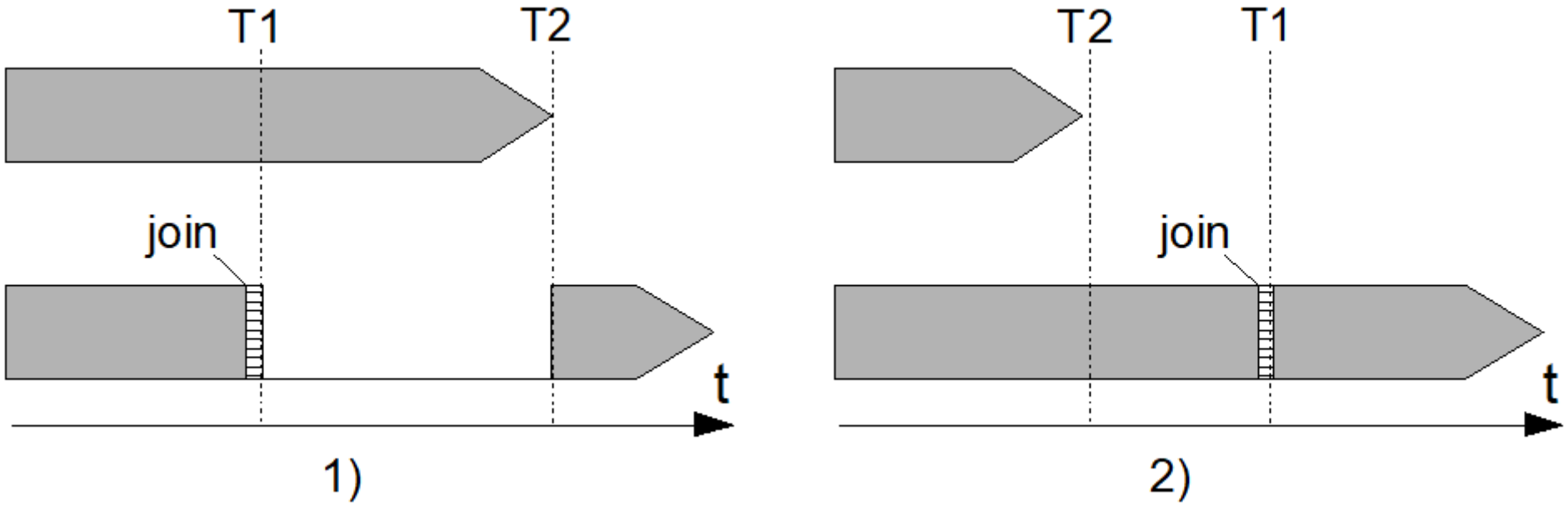
Priključivanje niti

- Ovakav model sinhronizacije, u kojem jedna nit čeka na završetak druge, naziva se **priključivanje niti** (engl. join).
- Na ovaj način, tok programa koji se izvršavao u niti koja se čeka, priključuje se toku programa niti koja čeka, i kaže se da se jedna nit priključuje drugoj.

```
void main() {
  std::thread thread1(print1);
  thread1.join(); // main će ovde čekati sve dok se thread1
                  // ne završi
} // thread1 se uništava (poziva se destruktor)
```

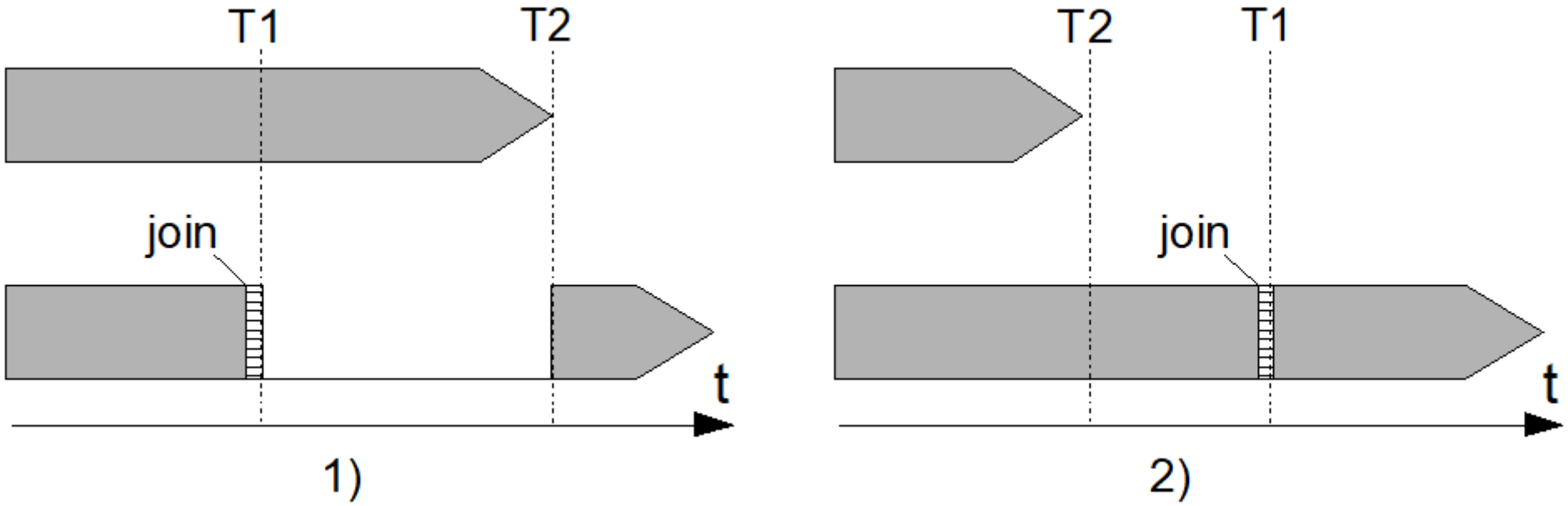
Priključivanje niti

- Ilustrovana su dva slučaja čekanja niti na priključivanje:
- U prvom slučaju nit kojoj se priključuje prelazi u stanje čekanja (trenutak T1) pre nego što je nit koja se priključuje završila svoj tok (trenutak T2).
- U drugom slučaju nit koja se priključuje završava sa radom (trenutak T2) pre nego što je nit koja priključuje spremna da je priključi (trenutak T1).



Priključivanje niti

- U prvom slučaju nit kojoj se priključuje stvarno čeka neko vreme.
- U drugom slučaju nit kojoj se priključuje ne čeka ništa. Međutim, nju u memoriji čeka informacija o završetku niti koja se priključuje.



Priključivanje niti

- Razradimo još dva scenarija priključivanja niti:
  1. Nit izavršava takvu funkciju da je ne treba priključivati.
  2. Novu nit ne želimo da priključimo iz niti (bloka koda) u kojoj je nastala, već na nekom drugom mestu.

Priključivanje niti

1. Nit izvršava takvu funkciju da je ne treba priključivati.

```
void foo() {
  // do something
}
void bar() {
  std::thread thread1(foo);
  thread1.detach(); // ovim saopštavamo da thread1 više
                    // nije priključiva nit
}
```

- Resursi neprikljućive niti će biti automatski oslobođeni kada se nit završi (jer informacija o njenom završetku se više nikoga ne tiče).
- U principu, nit može i da se nikad ne završi, ali to je već složenija diskusija.

Priključivanje niti

2. Novu niti ne želimo se da priključimo iz niti (bloka koda) u kojoj je nastala, već na nekom drugom mestu.

```
void foo();
std::thread bar() {
  std::thread thread1(foo);
  return thread1;
}
void main() {
  std::thread mainT{bar()};
  mainT.join();
}
```

```
void foo();
void bar(std::thread t) {
  // ...
  t.join();
}
void main() {
  std::thread thread1(foo);
  bar(std::move(thread1));
}
```

- std::thread se ponaša slično jedinstvenom pokazivaču.
- Ne mogu postojati dva objekta koji se odnose na istu nit!
- Ima samo move konstruktor i move operator dodele.

Prenošenje parametara funkciji niti, nastavak

- Kako se prenose parametri funkciji niti: po vrednosti, ili po referenci?

```
struct foo {
  void operator()(int x);
};

std::thread bar() {
  int i = 6;
  std::thread t(foo(), i);
  return t;
}
```

- Ovde je stvar jasna i nema nikakvih problema.

Prenošenje parametara funkciji niti, nastavak

- Kako se prenose parametri funkciji niti: po vrednosti, ili po referenci?

```
struct foo {
  void operator()(const int& x);
};

std::thread bar() {
  int i = 6;
  std::thread t(foo(), i);
  return t;
}
```

- Kada bi **i** prosledili po reference, nastao bi problem. Nit t nastavlja da živi i nakon što se funkcija bar završi, a sa završetkom bar, nestaje **i.**
- Zato se ovde zapravo i neće desiti prenošenje po referenci, tj. thread konstruktor pravi kopiju.

Prenošenje parametara funkciji niti, nastavak

- Ako zaista hoćemo po referenci, moramo pisati ovako:

```
struct foo {
  void operator()(const int& x);
};

std::thread bar() {
  int i = 6;
  std::thread t(foo(), std::cref(i)); // ref ako nemamo const
  return t;
}
```

- Ali tada moramo biti jako pažljivi.
- Nitske funkcije bi trebalo da uvek primaju parametre po vrednosti, osim u specijalnim slučajevima kada jasno znamo šta radimo.

Prenošenje parametara funkciji niti, nastavak

- Ova pravila važe i za lambda funkcije:

```
std::thread t([](int i){...}, i);
std::thread t([](int& i){...}, std::ref(i));
```

- Ali ne važe pri zahvatanju atributa:

```
std::thread t([i]() {...});
std::thread t([&i]() {...});
```

- U drugom slučaju će i biti stvarno zahvaćeno po referenci.

Dodatne korisne stvari

- std::thread ima još dve metode koje su nekada korisne:

```
std::thread t(foo());
t.get_id();   // vraća jedinstveni identifikator niti
t.joinable(); // proverava da li je nit priključiva
```

- Nekoliko korisnih stvari koje se zovu iz same niti:

```
void foo() {
  std::this_thread::get_id();
  ...
  std::this_thread::sleep_for(
      const chrono::duration<Rep, Period>& x);
  // primer upotrebe: std::this_thread::sleep_for(2s);
  ...
  std::this_thread::yield(); // hint operativnom sistemu da
}                            // prepusti procesor drugoj niti
```

Međusobna isključivost

- Za konkurentno (i paralelno) programiranje karakterističan je problem kritične sekcije.
- Kritična sekcija je deo koda niti koji pristupa deljenom resursu (kome neka druga nit takođe može da pristupi).
- Najbolje rešenje problema kritične sekcije jeste da se kod prepravi tako da ne pristupa deljenoj promenljivoj.
- Drugo rešenje je da se pristup deljenoj promenljivoj zaštiti objektom isključivog pristupa (muteksom).

```
std::mutex m;
void foo() {
  m.lock();
  ... // pristup deljenom resursu
      // samo jedna nit se može ovde nalaziti
  m.unlock();
}
```

Međusobna isključivost

- Muteks se ne može kopirati, niti premeštati (nema move-a).
- Metoda lock je blokirajuća: Ako je muteks već zaključan, nit će tu čekati dok se muteks ne oslobodi.
- Postoji i neblokirajuća varijanta:

```
std::mutex m;
void foo() {
  if (!m.try_lock()) return;
  ... // pristup deljenom resursu
      // samo jedna nit se može ovde nalaziti
  m.unlock();
}
```

Međusobna isključivost

- Muteks je resurs, koji se zauzima i oslobađa.
- Postoji identičan problem kao kod sirovih pokazivača koji su vlasnici dinamički alocirane memorije:

```
std::mutex m;
void foo() {
  m.lock();
  ... // pristup deljenom resursu
      // šta ako se desi izuzetak ovde?
  m.unlock(); // ovo možemo lako zaboraviti, pogotovo ako
              // je tok izvršavanja malo razgranatiji
}
```

Međusobna isključivost

- Muteks je resurs, koji se zauzima i oslobađa.
- Osnovni princip RAII (Resource Acquisition Is Initialization): „Vlasništvo“ nad resursom (objektom koji nema doseg) dodeliti nekoj promenljivoj koja ima doseg.

```
std::mutex m;
void foo() {
  std::lock_guard<std::mutex> lock(m); // muteks se zauzima
                                       // u konstruktoru
  ... // pristup deljenom resursu
} // u destruktoru se muteks oslobađa; radi i za izuzezke
```

Međusobna isključivost

- Kod konkurentnog programiranja, jedna od opasnosti je međusobno blokiranje procesa (engl. „deadlock“).
- Do toga može doći ukoliko niti moraju zauzeti više objekata isključivog pristupa (muteksa).
- Postoje razne tehnike za izbegavanje međusobnog blokiranja, a u okviru standardne biblioteke nudi se jedna tehnika koja očekuje da se svi potrebni muteksi zauzmu odjednom.

```
std::mutex m1, m2;
void foo() {
  std::scoped_lock lock(m1, m2); // ili više muteksa
  ... // pristup deljenom resursu
}
```

- Muteksi će biti zauzeti na način koji garantuje da neće doći do međusobnog blokiranja.

Međusobna isključivost

- std::scoped_lock je uveden od C++17
- U C++11/14 sličnu funkcionalnost je moguće ostvariti kombinovanjem std::lock() šablona sa promenljivim brojem argumenata i std::lock_guard.

```
std::mutex m1, m2;
void foo() {
  std::lock(m1, m2); // ili više muteksa
  std::lock_guard<std::mutex> lg1(m1, std::adopt_lock);
  std::lock_guard<std::mutex> lg2(m2, std::adopt_lock);
  ... // pristup deljenom resursu
}
```

- std::lock() zauzima dva ili više muteksa i tako izbegava međusobno blokiranje.
- Kada se std::adopt_lock prosledi konstruktoru klase std::lock_guard, neće se ponovo zauzimati muteks, a obezbeđujemo da se muteks oslobodi u destruktoru.

Međusobna isključivost

- Postoje još neke vrste muteksa:
- Vremenski ograničen muteks (std::timed_mutex)
  - Pruža funkcije za vremensko ograničeno neblokirajuće zaključavanje (try_lock_for i try_lock_until)
- Rekurzivni muteks (std::recursive_mutex)
  - Moguće ga je rekurzivno zaključavati iz iste niti.
  - Zbog toga je proces zaključavanja/otključavanja skuplji, pa koristiti ovo samo kada je potrebno.
  - Postoji i vremenski ograničen rekurzivni muteks.
- Deljeni muteks (std::shared_mutex)
  - Omogućava zaključavanje muteksa sa saopštavanjem namere.
  - Na taj način moguće je da više niti pristupa jednom deljenom resursu: jedna da piše, a više njih da čita.
  - Postoji i vremenski ograničen deljeni muteks.

Međusobna isključivost

- Interesantna je još i klasa unique_lock.
- To je najfleksibilnija brava.
- Omogućava vremenski ograničeno neblokirajuće zaključavanje (ukoliko ga muteks podržava)
- Omogućava rekurzivno zaključavanje (ukoliko ga i muteks podržava)
- Podržava premeštanje (move), ali ne i kopiranje.
- Omogućava zaključavanje/otključavanje i mimo konstruktora/destruktora.

Međusobna isključivost

- Ova mugućnost je vrlo važna

```
std::mutex m;
void foo() {
  std::unique_lock<std::mutex> lock(m);
  ... // pristup deljenom resursu
  lock.unlock();
  ... // sada je muteks (privremeno) oslobođen
  lock.lock();
  ... // a sada je opet zauzet
} // osobađanje muteksa u destruktoru
```

std::unique_lock

- Greška pri korišćenju unique_lock klase koju je lako napraviti, a teško uočiti:

```
std::mutex my_mutex;
void foo() {
  std::unique_lock<std::mutex> (my_mutex);
  // kao da smo napisali:
  // std::unique_lock<std::mutex> my_mutex();
  ... // pristup deljenom resursu
} // osobađanje muteksa u destruktoru, ali nema muteksa!
```

- Prevodilac na problematičnu liniju gleda kao na deklaraciju promenljive my_mutex tipa unique_lock korišćenjem podrazumevanog konstruktora.
- **Epilog - nismo zaključali ništa!**

Uslovne promenljive

- Često imamo potrebu za ovakvim kodom:

```
std::mutex m;

void foo() {
  std::unique_lock<std::mutex> lock(m);
  ...
  while (!some_condition);
  ... // radi nešto za šta je bio potreban uslov
}
```

- Jasno je da se u while petlju gubi vreme.

Uslovne promenljive

- Zato je ovo bolja varijanta.

```
std::mutex m;
std::condition_variable cv;

void foo() {
  std::unique_lock<std::mutex> lock(m);
  ...
  cv.wait(lock, [](){ return some_condition; });
  ... // radi nešto za šta je bio potreban uslov
}
```

Uslovne promenljive

- wait metodi se prosleđuje i brava, jer dok nit čeka, treba da otključa bravu, a čim se uslov zadovolji pa želi da nastavi sa radom, treba opet da je zaključa (zato mora de se koristi unique_lock).
- Metoda je blokirajuća i zato se ne gubi vreme koje ide na izvršavanje while petlje.
- Međutim, kada nit treba da se probudi i ponovo proveri uslov?

```
std::mutex m;
std::condition_variable cv;
void foo() {
  std::unique_lock<std::mutex> lock(m);
  ...
  cv.wait(lock, [](){ return some_condition; });
  ... // radi nešto za šta je bio potreban uslov
}
```

Uslovne promenljive

- Pa onda kad joj javi neka druga nit, koja možda baš promeni nešto što utiče na uslov.

```
std::mutex m;
std::condition_variable cv;

void foo() {
  std::unique_lock<std::mutex> lock(m);
  ...
  cv.wait(lock, [](){ return some_condition; });
  ... // radi nešto za šta je bio potreban uslov
}
void bar() {
  ... // nešto što utiče na some_condition
  cv.notify_one(); // ili notify_all()
}
```

Semafori

- Semafori su još jedan zgodan način sihronizacije, koji nije deo C++ standardne biblioteke, ali se lako pravi na osnovu uslovne promenljive.
- Formalno: semafor je ne-negativan broj S, nad kojim su definisane dve nedeljive operacije: V(signal) i P(wait).
- V(S) uvećava S za jedan.
- P(S) umanjuje S za jedan, ukoliko S nije 0.
- Za semafor kažemo da je signaliziranom stanju (tj da je signaliziran), ako je S različito od 0. U suprotnom je u nesignaliziranom stanju (nesignaliziran je) i na njemu se mora čekati.

Semafori

```
class Semaphore {
  int m_s = 0;
  std::mutex m_mut;
  std::condition_variable m_cv;
public:
  Semaphore() = default;
  Semaphore(int x) : m_s(x) {}
  void signal() {
    std::unique_lock<std::mutex> lock(m_mut);
    m_s += 1;
    m_cv.notify_one();
  }
  void wait() {
    std::unique_lock<std::mutex> lock(m_mut);
    m_cv.wait(lock, [this](){ return m_s != 0; });
    m_s -= 1;
  }
};
```

Semafori

- Semafori su zgodni za sihronizaciju oko kružnih bafera

```
class RingBuffer {
  std::array<int, 10> m_buff;
  int m_w{0};
  int m_r{0};
  Semaphore m_free{10};
  Semaphore m_taken{0};
  std::mutex m_mut;
public:
  RingBuffer() = default;
  void write(int x);
  int read();
};
```

Semafori

```
void RingBuffer::write(int x) {
  m_free.wait();
  std::lock_guard<std::mutex> l(m_mut);
  m_buff[m_w] = x;
  m_w = (m_w + 1) % 10;
  m_taken.signal();
}

int RingBuffer::read() {
  m_taken.wait();
  std::lock_guard<std::mutex> l(m_mut);
  int res = m_buff[m_r];
  m_r = (m_r + 1) % 10;
  m_free.signal();
  return res;
}
```

std::atomic

- Recimo da dve niti pristupaju samo jednoj deljenoj promenljivoj, koja predstavlja prost brojač koji obe niti po potrebi inkrementiraju.
- Jedina naredba koju obe niti izvršavaju nad brojačem je counter++.
- Laički možemo zaključiti da nije potrebno raditi sinhronizaciju.
- **Pogrešno!**
- Inkrementiranje promenljive se na nivou mašinskih instrukcija obavlja u tri koraka:
  - load (učitavanje iz memorije u registar)
  - add (inkrementiranje vrednosti u registru)
  - store (kopiranje iz registra u memoriju)
- Problem nastaje kada jedna nit bude prekinuta u sred inkrementiranja od strane druge niti koja takođe inkrementira deljeni brojač. Nakon što obe niti završe sa inkrementiranjem, brojač će pogrešno biti uvećan za 1 umesto za očekivanih 2.

std::atomic

- Jedno rešenje je da brojaču pridružimo muteks.
- Standardna biblioteka nam nudi jednostavnije i efikasnije rešenje, std::atomic:

```
std::atomic<int> counter{0}; // ili counter(0), ali
                             // ne može counter = 0
void foo() {
  ... // radi nešto
  counter++; // sada je bezbedno
}
```

- Garantovano nam je da će se inkrementiranje izvršiti atomično, posebnim instrukcijama ako ih platforma podržava ili softverski.

std::atomic

- Šta sve možemo raditi sa std::atomic promenljivima?
- Preklopljeni su operatori ++, --, +=, -=.
- Ista funkcionalnost se ostvaruje metodama: fetch_add i fetch_sub.
- Ako je u pitanju celobrojni tip, na raspolaganju su i operatori: &=, |= i ^=, kao i metode fetch_and, fetch_or i fetch_xor.

```
std::atomic<int> value{0};

void foo() {
  value.fetch_and(0xff); // ili value &= 0xff;
}
```

std::future

- U nekim situacijama nam je potrebno da nit vrati neku povratnu vrednost po obavljenom poslu.
- std::thread ne pruža takvu mogućnost.
- Jedno rešenje bi bilo korišćenjem globalnih promenljivih.
- Standardna biblioteka nam nudi std::future i std::async za dobijanje povratne vrednosti iz nitske funkcije.

```
int foo(int x);
int main() {
  std::future<int> rez = std::async(foo, i);
  std::cout << "Radi nesto dok se izvrsava foo" << std::endl;
  std::cout << "Rezultat = " << rez.get() << std::endl;
}
```

std::async

- std::async kao prvi argument može da primi i smernice za pokretanje („launch policy“).
- Opcije su:
  - std::launch::async – obaveza pokretanja u posebnoj niti
  - std::launch::deferred – pokretanje u istoj niti, tzv. „lazy evaluation“
  - std::launch::async | std::launch::deferred (default) - ostavlja se operativnom sistemu da odluči na koji način će pokrenuti.

```
int foo(int x);
int main() {
  auto rez = std::async(std::launch::async, foo, i);
  std::cout << "Radi nesto dok se izvrsava foo" << std::endl;
  std::cout << "Rezultat = " << rez.get() << std::endl;
}
```

std::async

- std::async je vrlo pogodan za paralelizovanje izvršavanja nekog taska.
- Primer operacije nad ogromnim kontejnerom:
  - Ukoliko redosled izvršavanja potrebne operacije nad pojedinačnim elementima nije striktno određen, kontejner možemo podeliti na nekoliko delova i prepustiti std::async funkcijama da obave operaciju nad delovima kontejnera.

```
std::vector<int> numbers;
auto lam = [](auto begin, auto end)
    { return std::max_element(begin, end); };
auto fut1 = std::async(std::launch::async, lam,
    numbers.begin(), numbers.begin() + numbers.size() / 2);
auto fut2 = std::async(std::launch::async, lam,
    numbers.begin() + numbers.size() / 2, numbers.end());

std::cout << std::min(*fut1.get(), *fut2.get()) << std::endl;
```

std::async

- Još jedna od prednosti std::async u odnosu na std::thread se odnosi na propagiranje izuzetaka.
- Ako izuzetak napusti funkciju std::thread objekta, pozvaće se terminate() handler.
- Sa druge strane, ako se dogodi izuzetak u funkciji koju izvršava std::async, biće propagiran u trenutku kada pozovemo std::future::get() metodu.

```
int foo() { throw 5; }

auto fut = std::async(foo);
try {
  auto x = fut.get();
} catch(int e) {
  std::cout << "Uhvacen " << e;
}
```

std::promise

- Još jedan način za razmenu podataka među nitima može biti realizovana koristeći klasu std::promise.
- Obezbeđuje prostor za smeštanje vrednosti (ili izuzetka) koja se kasnije asinhrono može dohvatiti pomoću std::future objekta koji se kreira na osnovu std::promise.

```
void foo(std::promise<int> x) {
  x.set_value(100);
}

std::promise<int> prom;
std::future<int> fut = prom.get_future();
std::thread t(foo, std::move(prom));
std::cout << fut.get();
t.join();  // wait for thread completion
```

RT-RK
COMPUTER BASED SYSTEMS

Contact us

RT-RK Institute for Computer Based Systems
Narodnog fronta 23a
21000 Novi Sad
Serbia

www.rt-rk.com
info@rt-rk.com