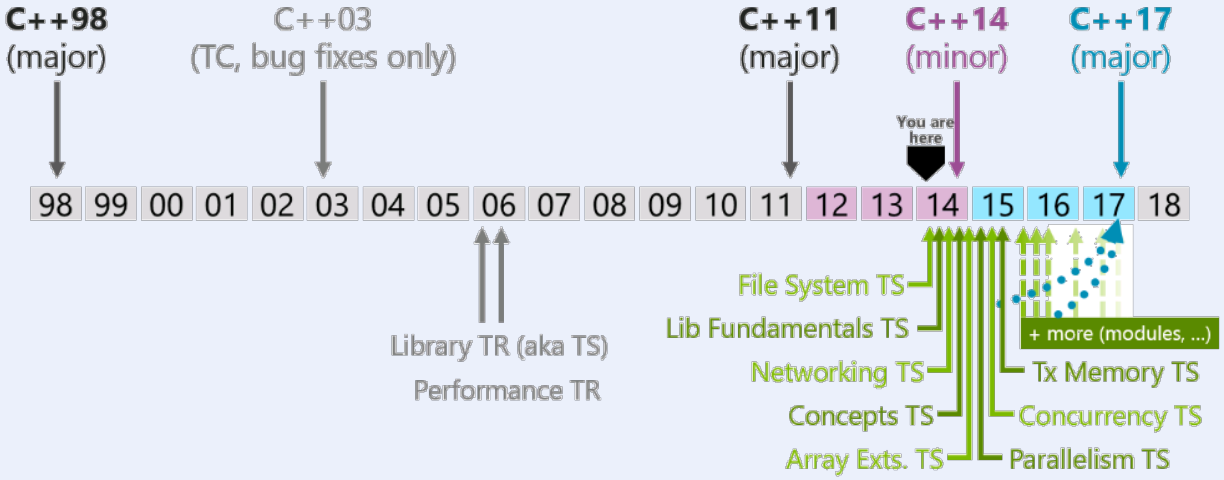
# Kaj je novega v C++11?
(in C++14)

Janez Brank

# Malo zgodovine




- Začetki C++ (prvotno "C with classes") segajo v leto 1979, večina njegovih značilnosti je bila razvitih v 80. letih
- S standardizacijo so začeli leta 1989
- C++98 – prvi ISO standard za C++
  - C++03 – manjši popravki C++98, relevantni le za pisce prevajalnikov
- C++11 – trenutni ISO standard za C++, mnogo novosti v primerjavi s C++98/03
  - Nekdaj znan tudi kot "C++0x"
- C++14 – prihajajoča majhna dopolnitev C++11
- C++17 – predvidena naslednja večja razširitev

Podpora C++11/14 v različnih prevajalnikih:
http://cpprocks.com/c1114-compiler-and-library-shootout/
http://clang.llvm.org/cxx_status.html
http://blogs.msdn.com/b/vcblog/archive/2014/06/11/c-11-14-feature-tables-for-visual-studio-14-ctp1.aspx

# Long long int

- V C++11 zdaj obstaja tudi **long long int** (in **unsigned long long int**)
  - Ki ga sicer mnogi prevajalniki že dolgo podpirajo
  - V C-ju je standarden že od C99
  - Vsaj tako dolg kot **long int**
  - Dolg vsaj 64 bitov

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari
  - V C++98 deklarira spremenljivko tipa T<…> po imenu z

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari
  - V C++98 deklarira spremenljivko tipa T<…> po imenu z
  - V C++11 primerja T<…>::x z globalno spremenljivko y, rezultat te primerjave pa z z

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari
  - V C++98 deklarira spremenljivko tipa T<…> po imenu z
  - V C++11 primerja T<…>::x z globalno spremenljivko y, rezultat te primerjave pa z z

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari
  - V C++98 deklarira spremenljivko tipa T<…> po imenu z
  - V C++11 primerja T<…>::x z globalno spremenljivko y, rezultat te primerjave pa z z

# Popravek pri sintaksi gnezdenih templateov

- V C++98 smo morali dva zaporedna znaka ">" ločiti s presledkom, sicer ju je prevajalnik obravnaval kot operator shift-right in se pritoževal:

```
vector<pair<int, int> > x; // veljavno v C++98 in C++11
```

- V C++11 to ni več potrebno:

```
vector<pair<int, int>> x; // veljavno v C++11, ekvivalentno gornjemu
```

  - "When parsing a template-argument-list, […] the first non-nested >> is treated as two consecutive but distinct > tokens" [C++11, sec. 14.2]

- Drobna nekompatibilnost zaradi tega:

```
template <int i> S { };
S<16>>2> x; // v C++98 deklarira x tipa S<4>; v C++11 je to sintaktična napaka
S<(16>>2)> y; // veljavno v obeh
```

- S primernimi deklaracijami se

```
T < U < r >> :: x > :: y > z;
```

  prevede v obeh jezikih in pomeni zelo različne stvari
  - V C++98 deklarira spremenljivko tipa T<…> po imenu z
  - V C++11 primerja T<…>::x z globalno spremenljivko y, rezultat te primerjave pa z z

# In-class member initializers

- Zdaj lahko napišemo

```
struct C {
    int foo = 10, bar;
    C() : bar(20) { }
    C(int _bar) : bar(_bar) { }
    C(int _foo, int _bar) : foo(_foo), bar(_bar) { }
};
```

- Kar je isto kot

```
struct C {
    int foo, bar;
    C() : foo(10), bar(20) { }
    C(int _bar) : foo(10), bar(_bar) { }
    C(int _foo, int _bar) : foo(_foo), bar(_bar) { }
};
```

- Prednosti tega postanejo bolj očitne, če imamo veliko kontruktorjev in veliko polj (memberjev), ki jih je treba inicializirati enako ne glede na to, kateri konstruktor je bil poklican

# Podedovani konstruktorji

- Tole nam gre najbrž vsem na živce:

```
struct Base {
   Base(int a, int b, ..., int z) { ... } };
struct Derived : public Base {
   Derived(int a, int b, ..., int z) : Base(a, b, ..., z) { }
};
```

- Včasih naš podrazred preprosto ne potrebuje nobene dodatne inicializacije in bi mu povsem ustrezali že konstruktorji nadrazreda
- Zdaj lahko naredimo takole:

```
struct Base {
   Base(int a, int b, int c) { } };
struct Derived : public Base {
   using Base::Base; // privleče Base-ove
                        konstruktorje v naš razred
};
Derived d(11, 12, 13); // OK, skonstruira d
     s konstruktorjem Base::Base(int, int, int)
```

- C++98 je podpiral nekaj podobnega za navadne metode, ne pa za konstruktorje

```
struct Base {
   void foo(int i);
   void bar(int i); };
struct Derived : public Base {
   using Base::bar;
   void foo(const char *p);  // skrije Base::foo
   void bar(const char *p);
};
Derived d;
d.foo("abc"); // OK, pokliče Derived::foo(const char *)
d.bar("abc"); // OK, pokliče Derived::bar(const char *)
d.foo(123); // napaka, 123 ni mogoče pretvoriti
                                          v const char *
d.bar(123); // OK, pokliče Base::bar(int)
```

# Delegiranje konstruktorjev

- Včasih več konstruktorjev dela (med drugim) iste stvari
  - Mogoče jih lahko združimo v en konstruktor z default parametri
    ```
    struct S { S(int a, int b, int c = –1) { … } };
    ```
  - Ali pa premaknemo skupno procesiranje v ločeno metodo
    ```
    class S {
        void init() { … }; // privatna metoda
    public:
        S(int a, int b) { … ; init(); }
        S(int a, int b, int c) { …; init(); } };
    ```
  - V C++11 imamo še eno možnost:
    konstruktor lahko kliče druge konstruktorje istega razreda
    ```
    struct S {
        S(int a, int b) { printf("a = %d, b = %d", a, b); }
        S(int a, int b, int c) : S(a, b) { printf(" in c = %d", c); }
    };
    S x(11, 12, 13); // izpiše "a = 11, b = 12 in c = 13"
    ```

# Null pointer literal

- **nullptr** je nova vrednost, ki predstavlja prazen kazalec in se ne pusti implicitno pretvoriti v **int**
  - Tip vrednosti **nullptr** je std::nullptr_t, ki ni pointer type,
    se pa pusti implicitno pretvoriti v poljuben pointer in v **bool**

```
char *p = nullptr, *q = 0;  // zdaj velja p == q
int *r = nullptr; // deluje za vsak tip kazalca
int s = nullptr; // napaka!
int t = (int) nullptr; bool b = nullptr; // OK, zdaj velja t == 0, b == false

void f(int);
void f(char *);
void g(int);

f(0);  // pokliče f(int)
f(nullptr); // pokliče f(char *)
g(nullptr); //napaka
```

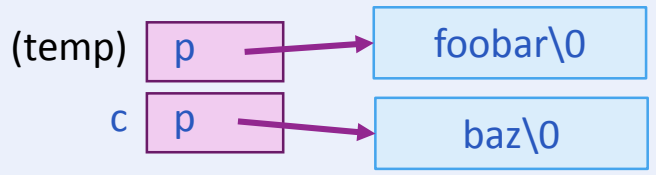
```
void h(char *);
void h(nullptr_t);

char *p = nullptr;
h(p);          // pokliče h(char*)
h(nullptr); // pokliče h(nullptr_t)
```

- V <cstddef> oz. <stddef.h> še vedno obstaja tudi NULL, ki pa ni priporočljiv
  - "The macro NULL is an implementation-defined C++ null pointer constant in this International Standard. Possible definitions include 0 and 0L, but not (**void***)0." [standard C++98, C++11, C++14]
    f(NULL); // pokliče f(**int**)!!!
  - "NULL which expands to an implementation-defined null pointer constant" [standard C99, C11]
  - To je zato, ker so v C++ pointerji strongly typed
    **char** *p = (**void** *) 0; // OK v C, napaka v C++

# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- V prireditvi c = a + b:
  - Pokliče se **operator** + (**const** S&, **const** S&) in vrne začasno instanco razreda S (ki vsebuje niz "foobar")
  - Pokliče se S::**operator** = (const S&) na instanci c in kot parameter dobi referenco na tisto začasno instanco; v c-ju alocira nov buffer in vanj skopira niz iz začasne instance

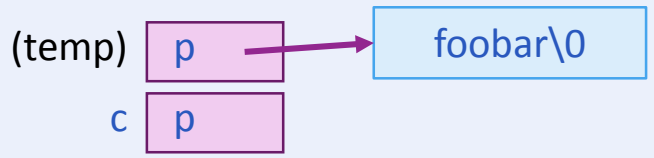
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- V prireditvi c = a + b:
  - Pokliče se **operator** + (**const** S&, **const** S&) in vrne začasno instanco razreda S (ki vsebuje niz "foobar")
  - Pokliče se S::**operator** = (const S&) na instanci c in kot parameter dobi referenco na tisto začasno instanco; v c-ju alocira nov buffer in vanj skopira niz iz začasne instance

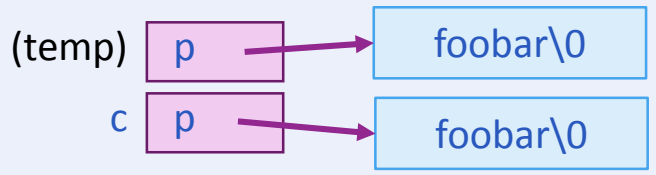
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- V prireditvi c = a + b:
  - Pokliče se **operator** + (**const** S&, **const** S&) in vrne začasno instanco razreda S (ki vsebuje niz "foobar")
  - Pokliče se S::**operator** = (const S&) na instanci c in kot parameter dobi referenco na tisto začasno instanco; v c-ju alocira nov buffer in vanj skopira niz iz začasne instance

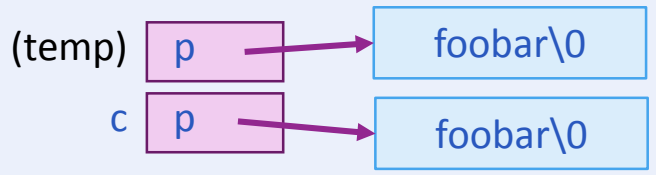
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- V prireditvi c = a + b:
  - Pokliče se **operator** + (**const** S&, **const** S&) in vrne začasno instanco razreda S (ki vsebuje niz "foobar")
  - Pokliče se S::**operator** = (const S&) na instanci c in kot parameter dobi referenco na tisto začasno instanco; v c-ju alocira nov buffer in vanj skopira niz iz začasne instance
  - Zdaj obstajata tako c kot začasna instanca, obe imata vsaka svojo kopijo niza "foobar"

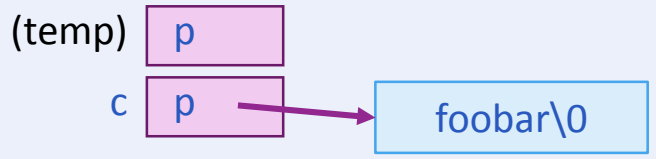
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- V prireditvi c = a + b:
  - Pokliče se **operator** + (**const** S&, **const** S&) in vrne začasno instanco razreda S (ki vsebuje niz "foobar")
  - Pokliče se S::**operator** = (const S&) na instanci c in kot parameter dobi referenco na tisto začasno instanco; v c-ju alocira nov buffer in vanj skopira niz iz začasne instance
  - Zdaj obstajata tako c kot začasna instanca, obe imata vsaka svojo kopijo niza "foobar"
  - Nato se kliče destruktor na začasni instanci
  - To je očitno potrata časa in prostora – bolje bi bilo, če bi lahko prireditev preprosto kanibalizirala začasno instanco, c bi posvojil njene podatke in jo pustil prazno

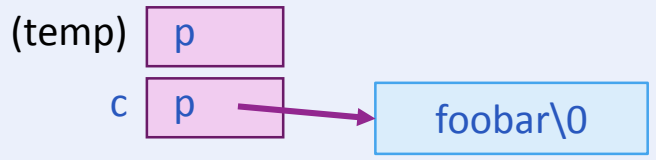
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja

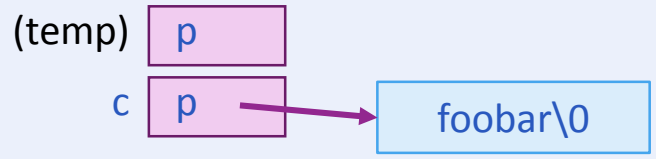
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Tradicionalno smo v C++ lahko definirali svoje copy konstruktorje in prireditvene operatorje
  - Pri nekaterih razredih je kopiranje drago (string, vector etc.)
  - Pogosto bi se kopiralo iz neke začasne instance, ki se jo bo kmalu za tem tako ali tako zavrglo
  - Dovolj bi bilo že premikanje, ki bi bilo cenejše od kopiranja
- Potencialno se lahko podobna neučinkovitost pojavi tudi v **operator +**
  - Najprej ustvari instanco razreda S v spremenljivki sum, kasneje pa stavek **return** to instanco skopira v začasno instanco, ki predstavlja rezultat (return value) funkcije
  - Prevajalniki so načeloma dovolj pametni, da to optimizirajo – sum ustvarijo kar na istem mestu, kjer se bo na koncu pričakovalo tisto začasno vrednost, ki jo vrača funkcija (*return value optimization*)
  - Ni pa to vedno mogoče, npr. če lahko funkcija vrača različne stvari
    - Npr. če ima `if (someCondition) return result1; else return result2;`

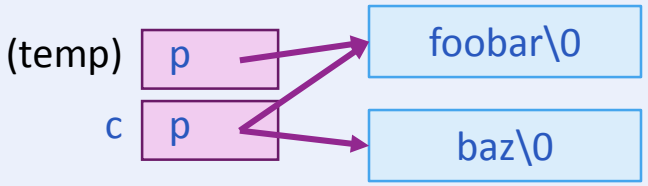
```
struct S {
   char *p;
   S() : p(nullptr) { }
   S(char *src) : p(new char[strlen(src) + 1]) {
                  strcpy(p, src); }
   ~S() { delete[] p; }
   S(const S& src) { p = new char[strlen(src.p) + 1];
                     strcpy(p, src.p); }
   S& operator = (const S& src) {
      if (this != &src) { delete[] p;
                  p = new char[strlen(src.p) + 1];
                  strcpy(p, src.p); }
      return *this; }
};
S operator + (const S& left, const S& right) {
   S sum; int n1 = strlen(left.p), n2 = strlen(right.p);
   sum.p = new char[n1 + n2 + 1];
   strcpy(sum.p, left.p); strcpy(sum.p + n1, right.p);
   return sum; }

S a = "foo", b = "bar", c = "baz";
c = a + b;
```



# Move semantics

- Zdaj lahko definiramo *move constructors* in *move assignment operators* poleg doslej običajnih *copy constructors* in *copy assignment operators*

```
struct S {
   ...
   S(S&& src) : p(src.p) { src.p = nullptr; } // move constructor
   S& operator = (S&& src) { // move assignment operator
      if (this != &src) { delete[] p;
         p = src.p; src.p = nullptr; }
      return *this; }
}
c = a + b;
```



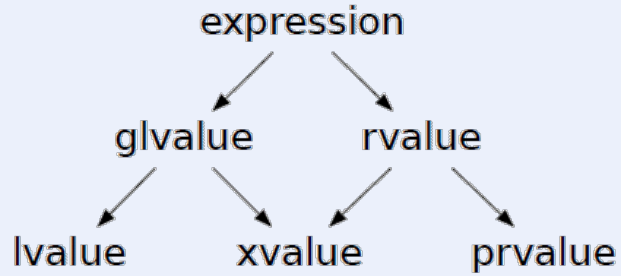

  - Zdaj se bo na c poklical S::**operator** =(S&&), kot parameter bo dobil referenco na začasno instanco, ki jo je vrnil **operator** +(**const** S&, **const** S&)
  - Move assignment operator ukrade p iz začasne instance in jo pusti prazno
  - Nato se kliče destruktor začasne instance (ki nima nobenega dela, ker je njen p takrat že **nullptr**)
  - Ta mehanizem bi se uporabil tudi pri vračanju instance S-ja iz klica funkcije ipd.
- Razredi iz standardne knjižnice (string, vector ipd.) imajo move konstruktorje in move assignment operatorje, dodali so tudi vector::push_back(S&&) in podobno
- Tehnično gledano je S&& *rvalue reference*, tradicionalna referenca oblike S& pa se zdaj imenuje *lvalue reference*
  - Lahko jih uporabljamo tudi drugod, ne le v konstruktorjih in prireditvenih operatorjih, vendar je to redko potrebno

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

```
struct S { … };
S bar() { … }
S x; bar() = x; // OK, kliče se S::operator=
// na začasni instanci, ki jo je vrnil bar()
// To deluje le za razrede:
int baz() { … }
baz() = 123; // napaka
```

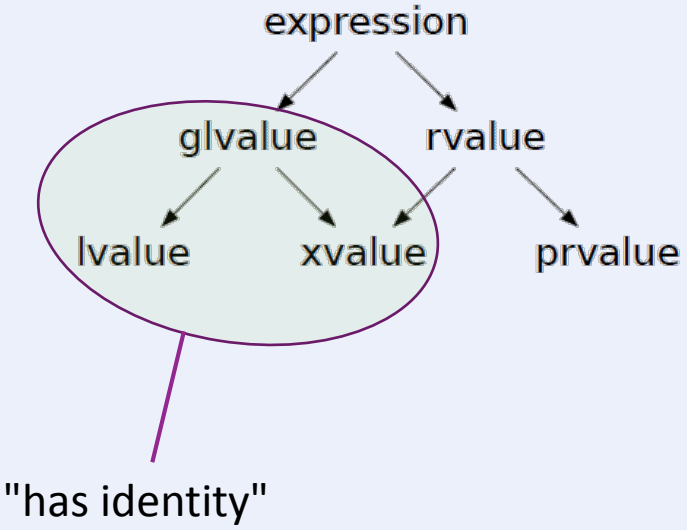
# Lvalues vs. Rvalues




- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

# Lvalues vs. Rvalues

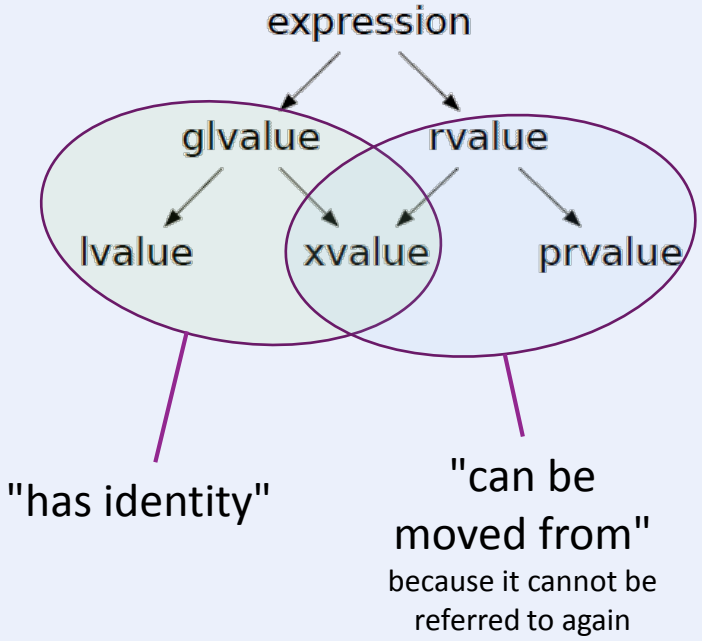
- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz




http://www.stroustrup.com/terminology.pdf

# Lvalues vs. Rvalues

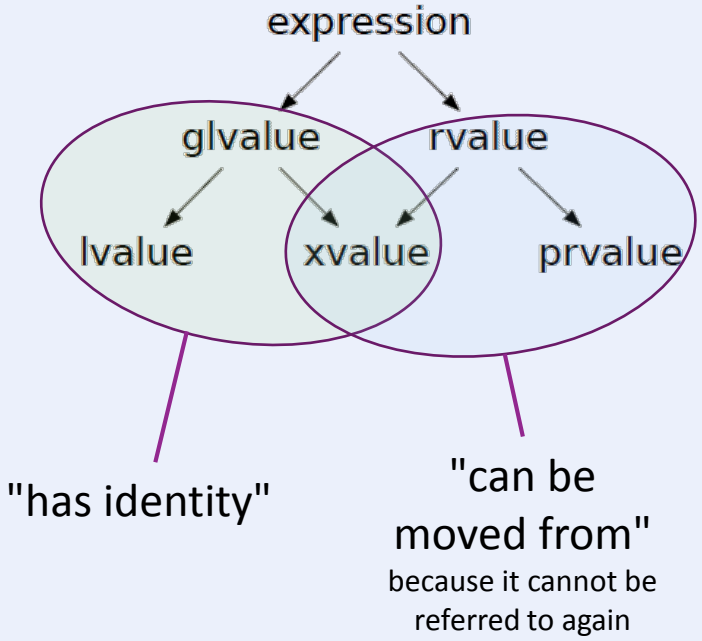
- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz




http://www.stroustrup.com/terminology.pdf

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

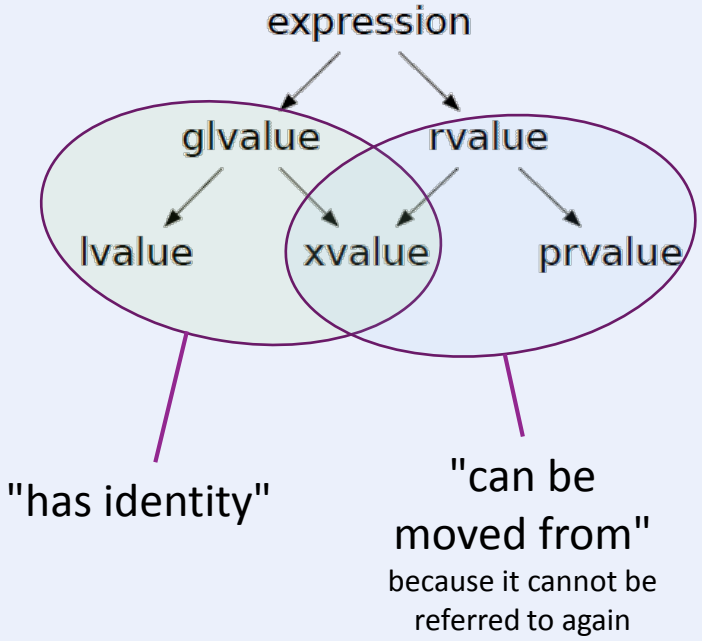



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12)  ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

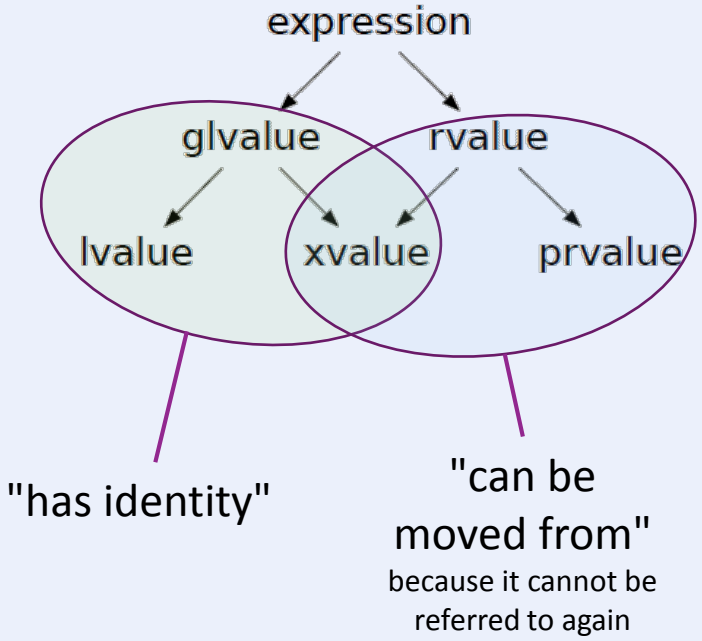



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12)  ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

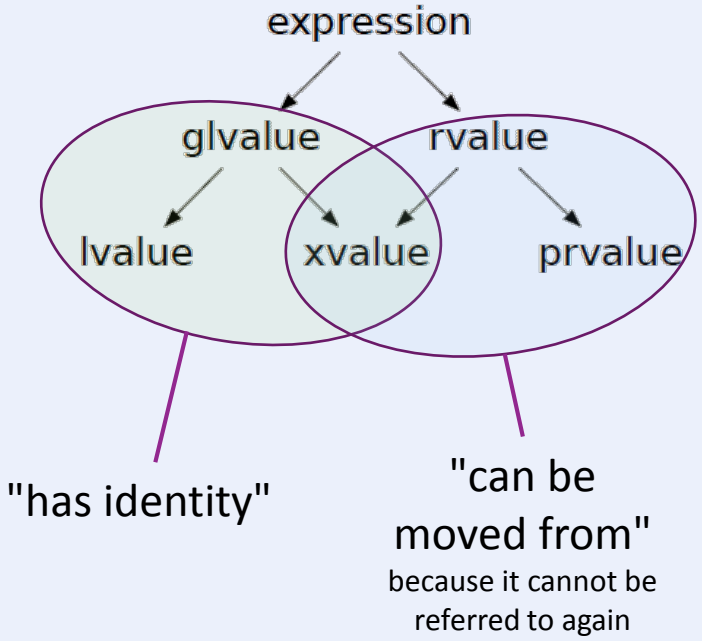



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12)  ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

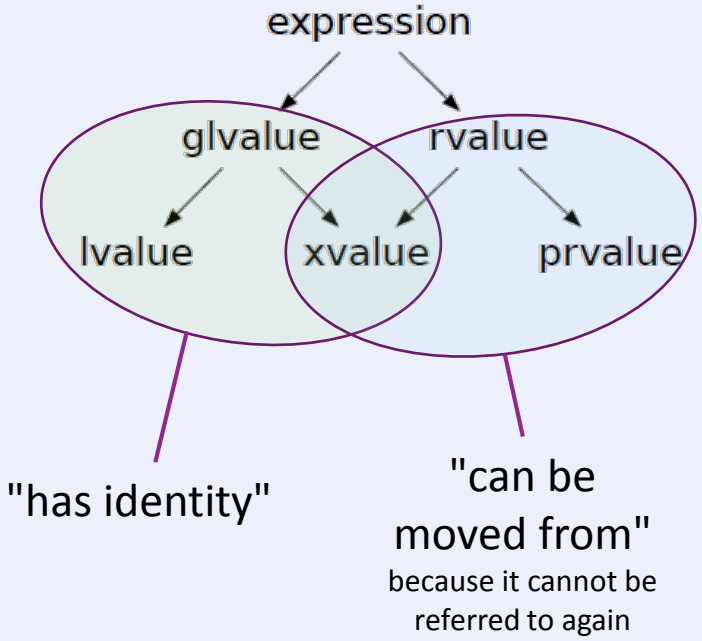



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12) ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

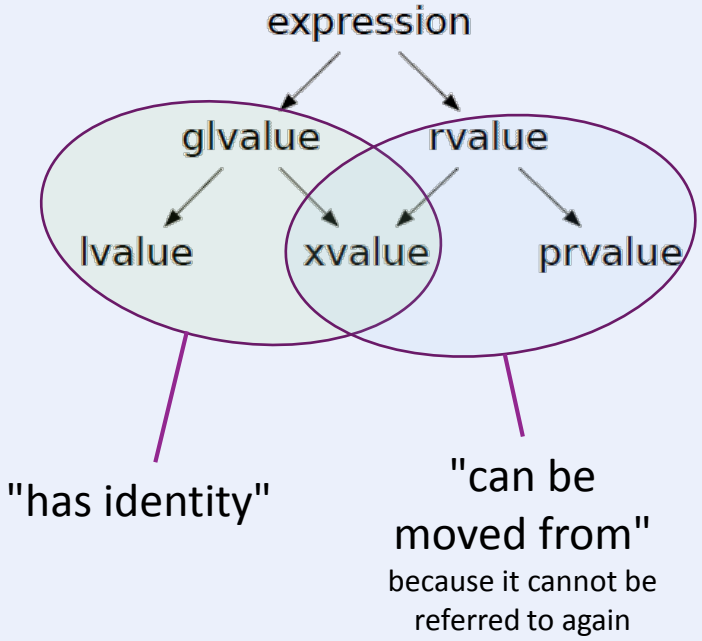



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12) ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

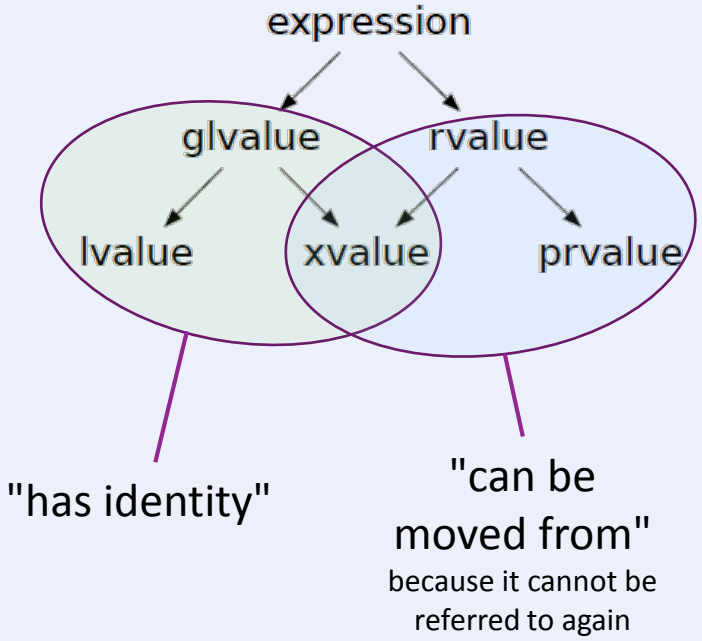



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12) ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz

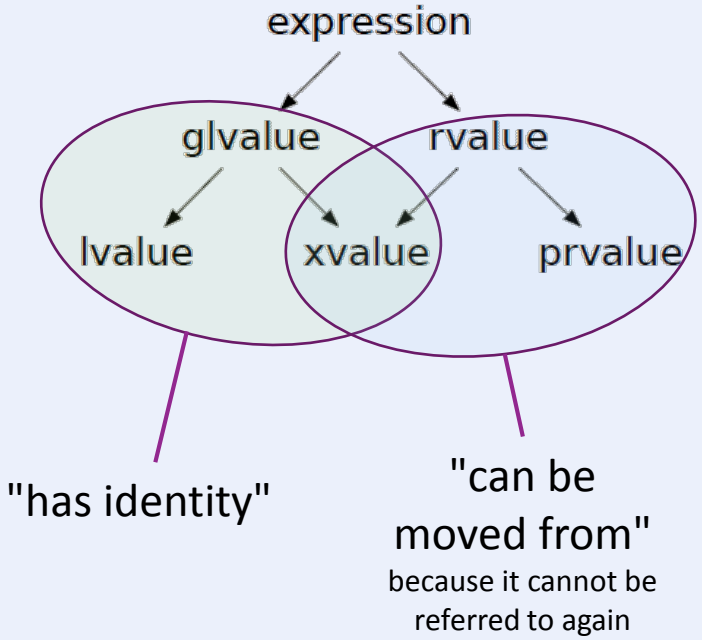



http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12) ;
```

# Lvalues vs. Rvalues

- Tradicionalni pogled:
  - Lvalue = izraz (*expression*), ki bi se lahko pojavil na levi strani prireditve
  - Rvalue = izraz, ki ni lvalue; lahko se pojavi le na desni strani prireditve
- Sodobnejši pogled:
  - Lvalue predstavlja neko funkcijo ali objekt; lahko vzamemo njegov naslov
    - Lahko je konstantnega tipa (in ne bi mogel biti na levi strani prireditve)
    - Če je izraz tipa lvalue reference (T& ali **const** T&), je ta izraz lvalue
  - Rvalue (= vsak izraz, ki ni lvalue) predstavlja
    - Začasen objekt (npr. rezultat casta ali klica funkcije, če ni tipa lvalue reference)
      - Ki se, mimogrede, prav lahko pojavi na levi strani prireditve
    - Ali pa sploh nima za sabo konkretnega objekta (npr. literal)
- Standard podrobno določa, kateri kategoriji pripada vsak možen izraz




http://www.stroustrup.com/terminology.pdf

```
int foo(int a, int b) { … };
int x, y, z;

x = y + foo(z, 12);
```

# Rvalue references

| Če imamo funkcijo... | ... ji lahko kot parameter podamo |
|---|---|
| **void** bar(**const** S&) | lvalue, const rvalue*, non-const rvalue[1] |
| **void** bar(S&&) | non-const rvalue[2] |

- Pravila za inicializacijo referenc
- Overload resolution,
  če imamo bar(**const** S&) in bar(S&&):
  - Če kot parameter podamo lvalue (ali const rvalue*),
    se kliče bar(**const** S&), saj se bar(S&&) niti ne bi mogla
  - Če kot parameter podamo (non-const*) rvalue,
    prevajalnik raje uporabi bar(S&&)

    *Ampak saj rvalues ponavadi niso const.
- To velja tudi pri konstruktorjih, prireditvenih operatorjih itd.

```
S foo() { ... }
void bar(const S&) { ... }
void bar(S&&) { ... }

S a = "aaa", b = "bbb";
bar(a);                 // pokliče bar(const S&) z referenco na a
bar("ccc");             // pokliče bar(S&&) z referenco na začasen objekt
bar(a + b);             // pokliče bar(S&&) z referenco na začasen objekt
bar(a = a + b);         // pokliče bar(const S&) z referenco na a
bar(foo());             // pokliče bar(S&&) z referenco na začasen objekt
bar(a = a + foo());     // pokliče bar(const S&) z referenco na a
```

[1] Visual Studio 2013 dovoli tudi non-const rvalue, a ne bi smel.

[2] Če ta rvalue še ne predstavlja konkretnega objekta, prevajalnik naredi začasen objekt in poda referenco nanj.

- Ta pravila veljajo tudi za inicializacijo spremenljivk tipa S&, **const** S&, S&& in **const** S&& in za vračanje rezultatov tega tipa iz klica funkcije.

# Še o premikanju

- Kaj pa, če hočemo neko lvalue obravnavati kot rvalue?

```
void Swap(S& a, S& b) {
    S temp = a;       // a je lvalue! pokliče se copy constructor
    a = b;            // b je lvalue! pokliče se copy assignment operator
    b = temp;         // temp je lvalue! pokliče se copy assignment operator
};
```

  - Lahko castamo:

```
S temp = (S&&) a; // izraz oblike "(S&&) a" je xvalue, kar je poseben primer rvalue
a = (S&&) b;
b = (S&&) temp;
```

  - Ali pa uporabimo std::move, ki v bistvu tudi ni nič drugega kot cast:

```
S temp = move(a); // move vrača S&&, zato je izraz "move(a)" tudi xvalue (in zato rvalue)
a = move(b);
b = move(temp);
```

  - Še en primer, ko je to pomembno: v move konstruktorjih in move assignment operatorjih

```
struct Inner { … };      // recimo, da ima tako move kot copy assignment operatorje
struct Outer {
    Inner x; char *p;
    Outer& operator= (Outer&& src) {
        x = src.x;           // src.x je lvalue! pokliče se Inner::operator=(const Inner&)
        p = src.p; src.p = nullptr; return *this; }
};
```

    - Zato moramo uporabiti x = move(src.x), da se pokliče Inner::**operator**=(Inner&&)

# Še o premikanju

| Če imamo… | Cena klica Set(lvalue) | Cena klica Set(rvalue) |
|---|---|---|
| Set(**const** Inner&) | Copy assignment | Copy assignment |

- Še en primer, ko pride premikanje prav: če hoče funkcija narediti kopijo argumenta

```
struct Inner { … };
struct Outer {
   Inner m;
   void Set(const Inner& src) { m = src; } // tako bi naredili v C++98
};
```

  - Outer::Set lahko kot src sprejme tako lvalue kot rvalue
  - V m = src pa uporabi copy assignment operator, saj move assignmenta ne more (src je lvalue)
- Če bi radi za src podali neko rvalue, bi bilo koristno, če bi znal Set uporabiti move asignment
  - Lahko bi dodali še void Set(Inner&& src) { m = move(src); }
  - Je pa neugodno, da potrebujemo 2 funkciji; če bi imeli *n* parametrov, bi potrebovali $2^n$ funkcij
- Še ena možnost je, da *namesto* prvotne Set uporabimo prenašanje po vrednosti!

```
void Set(Inner src) { m = move(src); }
```

  - Če je move konstruktor/assignment veliko cenejši od copy konstruktor/assignmenta, je ta rešitev praktično enako dobra kot prejšnja, zahteva pa le eno funkcijo namesto $2^n$ funkcij
  - To možnost je vredno imeti v mislih, ker smo se doslej prenašanja debelih objektov po vrednosti bali
  - Koristno tudi pri konstuktorjih: Outer::Outer(Inner _m) : m(move(_m)) { … }

# Ref-qualifiers

- Metoda S::foo(x, y, z) ima poleg eksplicitnih parametrov x, y, z še implicitni parameter – instanco razreda S, na kateri se jo kliče
  - Doslej smo lahko ločili metode po tem, ali je ta implicitni parameter tipa S& ali **const** S&

```
struct S {
    void foo() { … }
    void foo() const { … }
};
S x; const S y;
S bar() { return S(); }
const S cbar() { return S(); }
x.foo(); // pokliče S::foo()
y.foo(); // pokliče S::foo() const, saj S::foo() niti ne bi mogel
bar().foo(); // pokliče S::foo()
cbar().foo(); // pokliče S::foo() const
```

Posebno pravilo pri metodah brez ref-qualifierjev:

- Čeprav S::foo() vidi svoj implicitni parameter kot S&, se jo sme klicati tudi na rvalues
- In podobno za S::foo() **const**

  - V C++11 lahko ločimo metode tudi po tem, ali je ta implicitni parameter lvalue referenca ali rvalue referenca

```
struct S {
    void foo() & { … }
    void foo() const & { … }
    void foo() && { … }
    void foo() const && { … }
};
x.foo(); // pokliče S::foo() &
y.foo(); // pokliče S::foo() const &
bar().foo(); // pokliče S::foo() &&
cbar().foo(); // pokliče S::foo() const &&
```

- Metode z ref-qualifierjem & se sme klicati le na lvalues, tiste z ref-qualifierjem && pa le na rvalues
- Metod z in brez ref-qualifierji ne smemo mešati med sebo
  - Npr. imeti S::foo() in S::foo() &&

# Explicitly defaulted functions

- Vemo, da lahko prevajalnik implicitno pripravi nekaj posebnih metod našega razreda:
  - Default konstruktor (če nismo napisali nobenega svojega konstruktorja)
  - Destruktor (če nismo napisali svojega destruktorja)
  - Copy constructor, copy assignment operator
    (če ni nobenega copy/move konstruktorja ali copy assignment operatorja)
  - Move constructor, move assignment operator
    (če ni nobenega copy constructorja ali copy assignment operatorja)
- Zdaj lahko eksplicitno zahtevamo, naj prevajalnik pripravi kakšno od teh metod tudi takrat, ko je drugače (implicitno) ne bi

```
struct S1 {
    S1(int i) { }

};
S1 x; // napaka
```

```
struct S2 {
    S2(int i) { }
    S2() = default; // to sicer ni nič drugače kot S2() {}
};
S2 x; // OK
```

# Explicitly defaulted functions

- Včasih je lažje zahtevati "implicitno" verzijo funkcije kot pisati svojo:

```
struct S3 {
    S3() { }
    S3(S3&& src) { } // move constructor
};
S3 x; // OK, pokliče S3::S3()
S3 y = x; // napaka: x je lvalue, zato ne more uporabiti move konstruktorja;
    // in ker obstaja move konstruktor, prevajalnik ne bo implicitno naredil copy konstruktorja
```

- Zdaj lahko rečemo

```
struct S3 {
    S3() { }
    S3(const S3& src) = default; // zahtevajmo od prevajalnika še default copy konstruktor
    S3(S3&& src) { } // move constructor
};
S3 x; // OK, pokliče S3::S3()
S3 y = x; // OK, pokliče S3::S3(const S3&)
```

- Zakaj je bilo bolje zahtevati default copy konstruktor kot pisati svojega?
  - Zato, da se izognemo temu:

```
struct S3 {
   int a, b, c, ..., x, y, z;
   S3(const S3& src) : a(src.a), b(src.b), c(src.c), ..., x(src.x), y(src.y), z(src.z) { } // nadležno in zlahka se zmotimo
};
```

# Explicitly deleted functions

- Lahko eksplicitno pobrišemo funkcijo, ki bi jo sicer prevajalnik priskrbel implicitno

```
struct S1 {
   S(int i) { }

};
S1 x = 10;
S1 y = x; // OK
```

```
struct S2 {
   S(int i) { }
   S(const S&) = delete;
};
S2 x = 10;
S2 y = x; // napaka
```

  - V C++98 smo lahko enak učinek dosegli, če smo funkcijo le deklarirali, ne pa definirali, ali pa če smo konstruktor razglasili za privatnega

- To deluje tudi za druge metode, ne le konstruktorje

```
struct Base {
   int foo() { };
   int bar() { };
};
Base x;
x.foo(); // OK
x.bar(); // OK
```

```
struct Derived : public Base {

   int bar() = delete;
};
Derived y;
y.foo(); // OK
y.bar(); // napaka
((Base &) y).bar(); // OK
```

# Izvažanja templateov ni več

- Pri običajnih razredih:
  - Deklaracijo napišemo v header file, definicijo pa v .cpp file
  - Drugi .cpp fajli includajo samo header
- Pri templateih pa pišemo oboje v header file
  - Ker mora prevajalnik videti definicijo templatea, da lahko pripravi nove instanciacije, do katerih utegne priti, ko se bo ta template uporabljalo v drugih .cpp fajlih
  - V C++98 je obstajala rezervirana beseda **export**, s katero si lahko definicijo templatea pustil v enem samem .cpp fajlu in jo od tam "izvozil", da so jo uporabljali tudi drugi
  - To je bilo težko implementirati, uspelo je le enemu prevajalniku (Edison Design Group) in njegovi avtorji so predlagali, naj se **export** pobriše iz standarda
  - Tako je v C++11 **export** še vedno rezervirana beseda, a ne pomeni ničesar

# Template aliases z **using**

- Pogosto uporabimo **typedef**, da poimenujemo konkretno instanciacijo nekega templatea:

```
typedef pair<int, float> TIntFltPr;
TIntFltPr myPair;
```

- Včasih sem si želel, da bi bil tudi typedef lahko template:

```
template<typename T> typedef pair<T, T> TSamePair; // napaka
TSamePair<int> myPair;
```

- Načeloma bi lahko zavil typedef v nov template:

```
template<typename T> struct TSamePair {
   typedef TPair<T, T> type; };
TSamePair<int>::type myPair; // OK
```

- Zdaj pa lahko to storimo elegantneje z besedo **using**:

```
template<typename T> using TSamePair = pair<T, T>;
TSamePair<int> myPair;     // OK v C++11
```

- Deluje tudi drugod, ne le pri templateih:

```
typedef int myType;
using myType2 = int;          // ekvivalentno prejšnji vrstici
```

"We tried with the conventional and convoluted typedef solution, but never managed to get a complete and coherent solution until we settled on a less obscure syntax." – Stroustrup

# Dedukcija tipov z **auto**

- Pri deklaraciji spremenljivke lahko namesto tipa napišemo **auto**
- Prevajalnik bo določil tip na podlagi vrednosti izraza, s katerim spremenljivko inicializiramo

```
vector<pair<int, string>>::const_iterator i = v.begin(); // ugh
auto i = v.begin(); // mnogo lepše
```

- Lahko zahtevamo const ali referenco:

```
int foo();
auto x1 = foo(); // x1 je tipa int
auto &x2 = x1; // x2 je tipa int&
auto x3 = x2; // x3 je tipa int
const auto y1 = foo(); // y1 je tipa const int
auto &y2 = y1; // y2 je tipa const int&
auto y3 = y2; // y3 je tipa int
```

- Stari pomen besede **auto** kot storage specifierja ("to je običajna spremenljivka, ne statična ali eksterna") je zdaj prepovedan
- "This is the oldest C++0x feature; I implemented it in 1983, but was forced to take it out for C compatibility reasons." – Stroustrup
  - (Ker se je v C-ju takrat smelo deklarirati spremenljivko brez tipa in je to implicitno pomenilo, da je **int**)

# Dedukcija tipov z **decltype**

- **auto** pride prav, če hočemo le deklarirati spremenljivko, ki jo bomo takoj tudi inicializirali

```
template <typename T, typename U>
void mul(T t, U u) { auto product = t * u; }
```

- Kaj pa, če hočemo s tem tipom početi še kaj drugega?

```
template <typename T, typename U>
void mul(T *t, int len, U u) {
    typedef decltype(t[0] * u) Product;
    Product *products = new Product[len];
    for (int i = 0; i < len; i++) products[i] = t[i]* u; }
```

- Ali pa celo

```
template <typename T, typename U>
void mul(T t, U u, decltype(t * u)& result) { result = t * u; }
```

# Suffix return type syntax

- Kaj če bi hoteli iz mul neposredno vrniti rezultat?
```
template <typename T, typename U>
decltype(t * u) mul(T t, U u) { return t * u; } // napaka
```
- To ne deluje – ko prevajalnik pride do **decltype**(…), imen t in u še ne pozna
- Tole bi delovalo, vendar je grdo:
```
template <typename T, typename U>
decltype(*((T*)0) * *((U*)0)) mul(T t, U u) { return t * u; }
```
- Zdaj lahko tip rezultata funkcije navedemo tudi za parametri:
```
template <typename T, typename U>
auto mul(T t, U u) -> decltype(t * u) { return t * u; }
```
- To deluje tudi za ne-template funkcije
```
auto mul(int x, int y) -> int { return x * y; }
```
- V C++14 bo delovalo tudi brez sufiksa
```
auto mul(int x, int y) { return x * y; }
```
  - Prevajalnik iz stavka **return** ugotovi, kakšen tip vrača funkcija
  - Pri lambda funkcijah (ki jih bomo videli kasneje) je to možno že v C++11

# Lokalni tipi kot template argumenti

- Vedno se mi je zdelo rahlo hipokritsko, da C++ ne dovoli lokalnih funkcij, dovoli pa lokalne razrede:

```
void outer(int i) {
    void inner() { printf("hello"); }  // napaka, lokalna funkcija
    struct Inner {
        static void inner() { printf("hello"); } // OK
        static void inner2() { printf("%d", i); } // napaka, i ni deklariran
    };
    Inner::inner(); }
```

- Pred C++11 takšnih lokalnih tipov nismo mogli uporabiti kot template argumente, zdaj jih lahko

```
void foo(vector<int> & v) {
    struct MyComparator {
        bool operator()(int x, int y) { ... }};
    std::sort(v.begin(), v.end(), MyComparator()); }
```

# Lambde

- Lambda expressions so v bistvu syntactic sugar za definicijo preprostega lokalnega razreda:

```
int outer(int x) {
    auto sqr = [] (int y) { return y * y; };
    int z = sqr(x); return z + 10; }
```

- To je ekvivalentno takšnemu razredu:

```
int outer(int x) {
    class Inner {
        public: int operator()(int y) const { return y * y; }
    };
    Inner sqr;
    int z = sqr(x); return z + 10; }
```

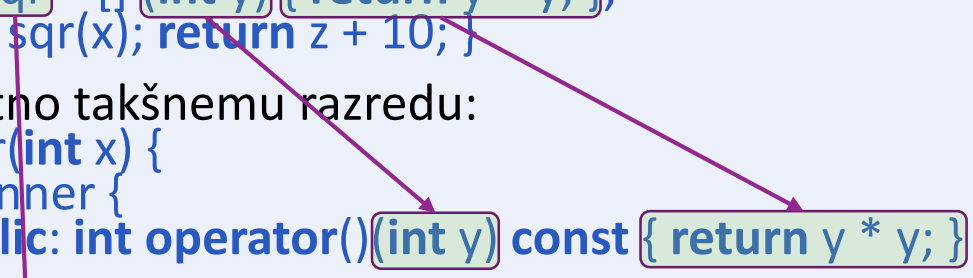
# Lambde

- Lambda expressions so v bistvu syntactic sugar za definicijo preprostega lokalnega razreda:

```
int outer(int x) {
    auto sqr = [] (int y) { return y * y; };
    int z = sqr(x); return z + 10; }
```

- To je ekvivalentno takšnemu razredu:

```
int outer(int x) {
    class Inner {
        public: int operator()(int y) const { return y * y; }
    };
    Inner sqr;
    int z = sqr(x); return z + 10; }
```



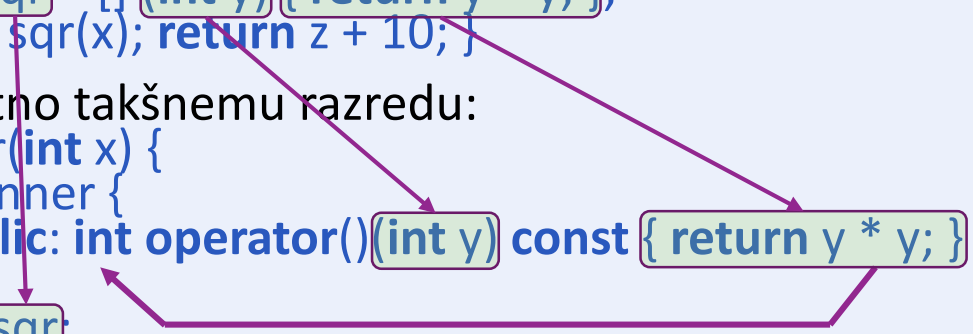
# Lambde

- Lambda expressions so v bistvu syntactic sugar za definicijo preprostega lokalnega razreda:

```
int outer(int x) {
    auto sqr = [] (int y) { return y * y; };
    int z = sqr(x); return z + 10; }
```

- To je ekvivalentno takšnemu razredu:

```
int outer(int x) {
    class Inner {
        public: int operator()(int y) const { return y * y; }
    };
    Inner sqr;
    int z = sqr(x); return z + 10; }
```



  - Prevajalnik je iz stavka "**return** y * y" ugotovil, da mora Inner::**operator**()(**int**) vračati **int**

# Lambde

- Lambda expressions so v bistvu syntactic sugar za definicijo preprostega lokalnega razreda:

```
int outer(int x) {
    auto sqr = [] (int y) { return y * y; };
    int z = sqr(x); return z + 10; }
```

- To je ekvivalentno takšnemu razredu:

```
int outer(int x) {
    class Inner {
        public: int operator()(int y) const { return y * y; }
    };
    Inner sqr;
    int z = sqr(x); return z + 10; }
```

  - Prevajalnik je iz stavka "**return** y * y" ugotovil, da mora Inner::**operator**()(**int**) vračati **int**
    - V C++14 bo to delovalo tudi, če bo več stavkov **return** (dokler vsi vračajo isti tip)
  - Lahko pa return type povemo tudi sami, s sufiksno sintakso:

```
double outer(int x) {
    auto sqr = [] (int y) -> double { return y * y; };
    double z = f(x); return z + 10; }
```

- V C++14 prihajajo generične lambde: kot če bi bil **operator()** template funkcija

```
auto sqr = [] (auto y) { return y * y; }
```

  - Kar je odstopanje od omejitve (ki bo še vedno veljala), da lokalni razredi ne smejo imeti template memberjev

# Lambde

- Ni nujno, da za lambdo deklariramo spremenljivko

```
vector<int> v;
sort(v.begin(), v.end(), [] (int x, int y) { return x > y; });
// uredi v padajočem vrstnem redu
```

- Lambda lahko počne vse, kar lahko počnejo običajne funkcije

```
vector<int> v = {19, 22, 30};
sort(v.begin(), v.end(), [] (int x, int y) {
    int sx = 0; while (x > 0) sx += x % 10, x /= 10;
    int sy = 0; while (y > 0) sy += y % 10, y /= 10;
    return sx < sy; }
); // uredi po vsoti števk: {30, 22, 91}
```

# Lambde in zajemanje lokalnih spremenljivk

- Enako kot vgnezden razred tudi lambda ne vidi avtomatsko lokalnih spremenljivk in parametrov zunanje fukcije

```
void outer(int x) {
   auto mulby = [] (int y) { return x * y; };  // napaka: x ni deklariran
   cout << mulby(x + 1); }
```

- Lahko pa prevajalniku naročimo, naj jih "zajame" (*capture*) :

```
void outer(int x) {
   auto mulby = [x] (int y) { return x * y; };  // OK
   return mulby(x + 1); }
```

- To je ekvivalentno takšnemu razredu:

```
void outer(int x) {
   class Inner {
      private: int x;
      public: Inner(int x_) : x(x_) { }
      int operator()(int y) const { return x * y; }
   };
   Inner mulby(x);
   return mulby(x + 1); }
```

- Vidimo, da ima lambda svojo kopijo x. Lahko pa x zajamemo po referenci:

```
auto mulby = [&x] (int y) { return x * y; };
```

  - Zdaj lahko lambda spreminja x in tudi vidi spremembe, ki jih na x povzroči outer

# Lambde in zajemanje

- Del lambde med oklepaji [ in ] je "*capture list*"
  - [] = ne zajemi ničesar
  - [=] = implicitno zajemi po vrednosti
    (vse lokalne spremenljivke, ki se omenjajo v lambdi)
  - [&] = implicitno zajemi po referenci
  - [a, b, &c, d, &e] = zajemi a, b, d po vrednosti, c in e pa po referenci
  - [=, &a, &b, &c] = zajemi a, b, c po referenci in vse ostalo implicitno po vrednosti
  - [&, a, b, c] = zajemi a, b, c po vrednosti in vse ostalo implicitno po referenci
  - V C++14 se bo dalo "zajeti" poljuben izraz
    (= z njim inicializirati memberja v lokalnem razredu, ki ga definira lambda)
    **auto** foo = [x = y + z, &u = v] (**int** a) { u += a; **return** x + a; }
    - Je isto kot

```
class Inner {
    private: int x, &u;
    public: Inner(int x_, int &u_) : x(x_), u(u_) { }
    int operator()(int a) { u += a; return x + a; }
};
Inner foo(y + z, v);
```

# Računanje ob prevajanju

- Nekatere stvari je res treba izračunati že med prevajanjem:

```
template <int len> class Arr { int a[len]; };
int mul(int a, int b) { return a * b; }

Arr<6> a1; // OK
Arr<2 * 3> a2; // OK
const int a = 2, b = 3; Arr<a * b> a3; // OK
Arr<mul(a, b)> a4; // napaka
```

- Toda človeku je očitno, da če poznaš a in b že ob prevajanju,
  bi lahko že takrat izračunal tudi mul(a, b)

```
constexpr int mul2(int a, int b) { return a * b; }

Arr<mul2(a, b)> a5; // OK
const int c6 = mul(2, 3);
Arr<c6> a6; // napaka: vrednost c6 ni znana ob prevajanju
const int c7 = mul2(2, 3);
Arr<c7> a7; // OK, vrednost c7 je znana ob prevajanju
constexpr int c8 = mul(2, 3); // napaka: mul ne moremo izračunati ob prevajanju
constexpr int c9 = mul2(2, 3); // OK; tudi mul2(a, b) bi bilo OK
Arr<c9> a9; // OK, c9 je constexpr

// mul2 deluje tudi kot normalna funkcija ob izvajanju programa
int x, y; cin >> x >> y; cout << mul2(x, y);
```

# Računanje ob prevajanju

- Omejitev za **constexpr** funkcije:
  - Telo funkcije mora obsegati le 1 stavek **return** in ničesar drugega
  - Še vedno imamo rekurzijo in ?: za vejitve, tako da lahko naredimo karkoli

```
constexpr int pow(int a, int b) {
    return b == 0 ? 1 : a * pow(a, b – 1); // OK, mogoče rahlo perverzno
}
```

  - To omejitev bodo v C++14 močno ublažili
    - Funkcija bo lahko imela vse razen **goto**, **asm**, **try**/**except**
    - Lahko bo imela spremenljivke, a morajo biti inicializirane, ne-statične, njihov tip mora biti literal
- Mimogrede, to ni tako dramatično, kot je mogoče videti na prvi pogled
- V C++98 bi lahko za enak namen kot tu **constexpr** funkcije uporabili template

```
template<int a, int b> Pow;
template<bool c, int a, int b> Helper {
    enum { value = 1 }; };          // to bomo uporabili le, ko bo c enak false
template<int a, int b> Helper<true, a, b> {
    enum { value = a * Pow<a, b – 1>::value }; };
template<int a, int b> Pow {
    enum { value = Helper<b == 0, a, b>::value }; };
// Zdaj je Pow<a, b>::value enaka vrednosti pow(a, b) in je tudi znana že ob prevajanju.
```

# Računanje ob prevajanju

- Omejitev za **constexpr** funkcije:
  - Telo funkcije mora obsegati le 1 stavek **return** in ničesar drugega
  - Še vedno imamo rekurzijo in ?: za vejitve, tako da lahko naredimo karkoli

```
constexpr int pow(int a, int b) {
    return b == 0 ? 1 : a * pow(a, b – 1); // OK, mogoče rahlo perverzno
}
```

  - To omejitev bodo v C++14 močno ublažili
    - Funkcija bo lahko imela vse razen **goto**, **asm**, **try**/**except**
    - Lahko bo imela spremenljivke, a morajo biti inicializirane, ne-statične, njihov tip mora biti literal
  - Prevajalnik sme postaviti omejitve na največjo globino rekurzije (priporoča se vsaj 512 nivojev)
    - Več kot to se bo računalo ob izvajanju, če kontekst to dopušča

```
constexpr int identity(int x) { return x == 0 ? 0 : 1 + identity(x – 1); }
template <int i> class T { };
T<identity(500)> t1; // OK, t1 je tipa T<500>
T<identity(10000)> t2; // zelo verjetno compile time error
int y1 = identity(500); // OK, najbrž se izračuna že med prevajanjem
int y2 = identity(10000); // OK, najbrž se izračuna šele med izvajanjem
int y3 = identity(1000000000); // se prevede, najbrž doživi stack overflow med izvajanjem
```

# Compile-time assertions

- Uporabimo lahko kakršen koli pogoj, čigar vrednost je znana že ob prevajanju

```
int mul(int a, int b) { return a * b; }
constexpr int mul2(int a, int b) { return a * b; }

static_assert(mul(2, 3) > 5, "foo"); // napaka, mul ni constexpr
static_assert(mul2(2, 3) > 5, "foo"); // OK
const int a = 2, b = 3;
static_assert(mul2(a, b) > 5, "foo"); // OK
static_assert(mul2(a, b) > 7, "foo"); // napaka foo (assertion failed)
```

- Sintaktično gledano je **static_assert** deklaracija,
  zato jo lahko uporabljamo tudi zunaj funkcij
- Načeloma je tudi to le syntactic sugar – v C++98 bi lahko podobno dosegli takole

```
template<bool b> struct AssertClass { };
template<> struct AssertClass<false> { private: AssertClass() { } };
#define MyStaticAssert(b) AssertClass<(b)>()
```

# Strongly typed enums

- Glavne razlike med starimi in novimi enumi (naštevnimi tipi):
  - Imena iz starega enuma se vidi tudi zunaj njega (v njegovem scopeu), pri novem enumu se jih ne
  - Stare enume se lahko implicitno pretvori v int, novih se ne da
  - Pri novih enumih lahko povemo, na katerem celoštevilskem tipu naj temeljijo

```
typedef enum { Old1, Old2, Old3 } OldEnum;

OldEnum x1 = Old1; // OK
OldEnum x2 = OldEnum::Old1; // OK
OldEnum x3 = 1; // napaka, int → enum ne gre
OldEnum x4 = (OldEnum) 1; // OK
int x5 = Old1; // OK
```

```
typedef enum { Old1, Old2, Old3 } OldEnum2;
// napaka, redefinicija Old1 itd.
```

```
enum class NewEnum { New1, New2, New3 };

NewEnum y1 = New1; // napaka, New1 ni znan
NewEnum y2 = NewEnum::New1; // OK
NewEnum y3 = 1; // napaka , int → enum ne gre
NewEnum y4 = (NewEnum) 1; // OK
int y5 = NewEnum::New1; // napaka, NewEnum → int ne gre
int y6 = (int) NewEnum::New1; // OK
```

```
enum class NewEnum2 { New1, New2, New3 };
// OK, NewEnum2::New1 se ne tepe z NewEnum::New1

enum class NewEnum3 : char { New1, New2, New3 };
// sizeof(NewEnum) == 4, sizeof(NewEnum3) == 1
```

# Override keyword

- Oglejmo si tale dva razreda:

```
struct Base {
   virtual void f();
   virtual void g();
   void h();
};
struct Derived : public Base {
   void f();           // povozimo Base::f
   void g() const;  // to ni override, ker ne obstaja Base::g() const
   void h();           // to ni override, ker Base::h ni virtualna
};
```

- To je veljavno tako v C++98 kot v C++11,
  vendar Derived::g in Derived::h najbrž nista to, kar smo imeli v mislih
- V C++11 lahko s ključno besedo **override** rečemo prevajalniku
  "Tole naj bi povozilo neko virtualno metodo nadrazreda. Če je ne, javi napako"

```
struct Derived : public Base {
   void f() override;          // OK, povozimo Base::f
   void g() const override; // napaka
   void h() override;          // napaka
};
```

# Final keyword

- Podobno kot v javi ipd. lahko s ključno besedo "**final**" preprečimo dedovanje:

```
struct Base final { ... };
struct Derived : public Base { ... };  // napaka, Base je final
```

- Tudi pri posameznih metodah:

```
struct Base {
   virtual void f() final;
   virtual void g();
};
struct Derived : public Base {
   void f(); // napaka, Base::f() je final
   void g(); // OK, povozimo Base::g()
};
```

# Nova sintaksa za inicializacijo

- C++ova obstoječa sintaksa za inicializacijo ima nekaj slabosti:

```
class C { … }
int foo = 10; class bar { … };
C a(10), b(foo);  // OK, skonstruira s C::C(int)
C a = C(10), b = C(foo); // enako
C a = C(), b; // OK, skonstruira s C::C()
C a(); // deklarira (in ne pokliče) funkcijo (ne spremenljivke)
C a(foo); // OK, skonstruira s C::C(int)
C a(bar); // deklarira (in ne pokliče) funkcijo (ne spremenljivke)
```

- Sintaksa za inicializacijo z { }, ki smo jo doslej poznali za arraye in strukture, deluje zdaj tudi drugod:

```
C a{10}, b{foo}; // OK, skonstruira s C::C(int)
C a = {10}, b = {foo}; // enako
C a = C{10}, b = C{foo}; // enako
C a{}, b = {}, c = C{}; // OK, skonstruira s C::C()
```

# Nova sintaksa za inicializacijo

- Deluje tudi drugod:

```
vector<pair<TFoo, TBar>> v;
v.push_back(pair<TFoo, TBar>(myFoo, myBar)); // ali gre to tudi vam na živce?
v.push_back({myFoo, myBar}); // precej lepše
```

- Dokler seveda stvari ne postanejo dvoumne:

```
struct C1 { C1(int, float) { } };
struct C2 { C2(int, float) { } };
void foo(C1 c1) { }
void foo(C2 c2) { }
foo({10, 3.14});  // napaka: ste mislili foo(C1) ali foo(C2)?
foo(C1{10, 3.14}); // OK
```

# Initializer lists

- Svoj razred lahko inicializiramo iz takšnega seznama:

```
template <typename T> struct TMyVec {
   T *vals; int len;
   template <typename U> TMyVec(std::initializer_list<U> lst) {
      len = 0; vals = new T[lst.len];
      for (auto p = lst.begin(); p != lst.end(); ++p) vals[len++] = *p; }
};
TMyVec<int> v = { 2, 10, 7, 3, 5};
```

- Deluje tudi za druge funkcije, ne le konstruktorje

```
void Print(initializer_list<int> lst) {
   printf("%d ints:", (int) lst.size());
   for (auto p = lst.begin(); p != lst.end(); ++p) printf(" %d", *p);
   printf("\n");
}
Print({10, 20, 30, 40});  // izpiše "4 ints: 10 20 30 40"
```

# Initializer lists

- Če ima razred tako "tradicionalne" konstruktorje kot take z initializer_list
  - T foo(…) lahko uporabi le tradicionalne konstruktorje
  - T foo{…} lahko uporabi oboje, vendar ima raje tiste z initializer_list, ČE lahko naredi homogen initializer_list
  - T foo = {…} je isto kot T foo {…}
- Primer:

```
struct T {
    T(int, int) { }
    T(int, double) { }
    T(initializer_list<int>) { }
    T(initializer_list<double>) { }
};
```

  - T foo(10, 20); // uporabi T::T(**int**, **int**)
    T foo{10, 20}; // uporabi T::T(initializer_list<**int**>)
  - T foo(10, 20.5); // uporabi T::T(**int**, **double**)
    T foo{10, 20.5}; // VC uporabi T::T(**int**, **double**)! ker seznam ni homogen; g++ javi napako
  - T foo(10.5, 20.5); // uporabi T::T(**int**, **double**)! ker s to sintakso ne more uporabiti konstruktorja z initializer_list
    T foo{10.5, 20.5}; // uporabi T::T(initializer_list<**double**>)
  - T foo(10, 20, 30); // napaka: noben konstruktor ne sprejema 3 argumentov
    T foo{10, 20, 30}; // uporabi T::T(initializer_list<**int**>)
  - T foo(10, 20, 30.5); // napaka: noben konstruktor ne sprejema 3 argumentov
    T foo{10, 20, 30.5}; // napaka, ker ne more sestaviti homogenega initializer_lista
  - T foo(initializer_list<double>{10, 20, 30.5}); // uporabi T::T(initializer_list<**double**>)
    T foo = initializer_list<double>{10, 20, 30.5}; // uporabi T::T(initializer_list<**double**>)
    T foo { initializer_list<double>{10, 20, 30.5}}; // skonstruira začasno instanco s T::T(initializer_list<**double**>)
    // in jo premakne v foo s T::T(T&&)

# Eksplicitni operatorji za pretvorbo tipov

- Konstruktorji omogočajo implicitne pretvorbe tipov:

```
struct S {
    S(int i) { … }
};
void foo(S s) { … }
S a = 12; foo(13); // oboje OK
```

- Če hočemo, lahko z besedo **explicit** to preprečimo:

```
struct S {
    explicit S(int i) { … }
};
void foo(S s) { … }
S a = 12; foo(13); // oboje sta napaki
S b(14), c{15}, d = S(16); // vse OK
```

- To je obstajalo že v C++98; v C++11 lahko enako naredimo tudi z operatorji za pretvorbo tipov:

```
struct S {
    operator int() { … }
};
void bar(int i) { … }
S a; int j = a; bar(a); // oboje OK
```

- Z **explicit**:

```
struct S {
    explicit operator int() { … }
};
void bar(int i) { … }
S a; int j = a; bar(a); // oboje sta napaki
int x(a), y{a}, z = (int) a, w = int(a); // OK
```

# Variadic templates

- Tradicionalni C-jevski pristop k funkcijam s spremenljivim številom argumentov:

```
void foo(int n, …) { // pričakuje še n parametrov tipa T
    va_list p; va_start(p, n);
    while (n-- > 0) {
        T t = va_arg(p, T);
        doSomethingWith(t); }
    va_end(p); }
```

- To se ne obnese, če je T nek nontrivially copyable class:

```
struct T {
    T() { }
    T(const T& src) { }
    T& operator = (const T& src) { return *this; }
    ~T() { }
};
T t1, t2, t3;
foo(3, t1, t2, t3);
```

  - Standard ne predpisuje, kaj naj se tu zgodi; prevajalnik bo mogoče javil napako
  - Ali pa bo skopiral t1, t2, t3 na sklad byte po byte in ignoriral naše copy konstruktorje in prireditvene operatorje
    - Prireditev T t = va_arg(p, t) pa bo inicializirala t s copy konstruktorjem T::T(**const** T& src), ki bo kot src dobil referenco na eno od tistih surovih kopij na skladu

# Variadic templates

- Zdaj imamo za to spodoben C++ovski mehanizem

```
int Output() { return 0; }

template <typename First, typename... Rest>
int Output(First first, Rest... rest) {
    cout << first;
    if (sizeof...(Rest) > 0) cout << ", ";
    return 1 + Output(rest...); }
```

  - Temu bi zlahka dodali še format string in bi imeli C++ovski ekvivalent printf
  - Kako to deluje?
    - rest je *parameter pack* – predstavljajmo si, da se v njem skriva $rest_1$, $rest_2$, …, $rest_n$
    - Izraz oblike "*f*(rest)…" je *pack expansion*, pri čemer je *f*(rest) njegov vzorec (*pattern*)
    - Ta izraz se spremeni v $f(rest_1), f(rest_2), \ldots, f(rest_n)$

```
int foo(int x) { return x + 1; }
void bar(int x, int y, int z) { printf("%d %d %d", x, y, z); }
template <typename... Args> void baz(Args... args) {
    bar((foo(args) + args)...);
}
baz(100, 1000, 10000); // izpiše 201, 2001, 20001
```

  - Variadic templates so možni tudi pri razredih, ne le funkcijah – koristno za nehomogene n-terice

```
std::tuple<string, int, double> t1 { "foo", 1, 2.3 };
auto t2 = make_tuple(string("foo"), 1, 2.3); // istega tipa kot t1
```

Instanciacija funkcije baz, ki se zaredi ob tem klicu, je videti takole:

```
void baz(int x, int y, int z) {
    bar(foo(x) + x, foo(y) + y, foo(z) + z);
}
```

# Template spremenljivke

- Doslej je bila template lahko le funkcija ali razred, od C++14 naprej tudi spremenljivka

```
template <typename T>
T pi = T(3.14159265358979323385);

template <typename T>
T area(T r) { return pi<T> * r * r; }
```

- Podobne reči so možne že v C++98, le oviti jo moramo v razred

```
template <typename T>
struct Pi { static T value; };

template <typename T>
T Pi<T>::value = T(3.14159265358979323385);

template <typename T>
T area(T r) { return Pi<T>::value * r * r; }
```

# User-defined literals

- Lahko definiramo funkcijo in prevajalnik jo bo poklical, če kak literal uporabi ime te funkcije kot sufiks:

```
string operator "" _rev(const char *p, size_t len) {
    string s(len, ' ');
    for (size_t i = 0; i < len; i++) s[i] = p[len – 1 – i];
    return s; }
string s = "abc"_rev; // pomembno: ni presledka med " in _
assert("cba" == s); // OK
```

  - Zakaj dobimo len? Ker se v literalu sme pojaviti tudi \0
- Deluje tudi za druge literale (številske, znakovne):

```
long double operator "" _ft(long double x) { return x * 0.3048; }
cout << 100.0_ft; // izpiše 30.48
```

  - Lahko tudi dobimo surovi int/float literal kot niz

```
int operator "" _sum(const char *p) {
    int n = 0; while (*p) n += *p++ – '0';   return n; }
int x = 11111111111111111111111111111_sum; // zdaj je x == 30
```

    - Tako bi se lahko definiralo nekakšen BigInteger literal

# User-defined literals

- Lahko celo dobimo posamezne znake literala kot parametre variadičnega templatea:

```
constexpr int pow3(int i) { // računa potence števila 3 ob prevajanju
   return i == 0 ? 1 : 3 * pow3(i – 1); }

template<char c, char... rest> struct helper {
   enum { value = (c – '0') * pow3(sizeof...(rest)) + helper<rest...>::value }; };
template<char c> struct helper<c1> {
   enum { value = c – '0' }; };

template<char... rest> constexpr int operator "" _ternary() {
   return helper<rest...>::value; }

cout << 212_ternary; // izpiše 21 (= 2 · 3^2 + 1 · 3^1 + 2 · 3^0)
```

- User-defined literali se morajo začeti na _ , tisti brez _ so rezervirani za standard
  - V C++11 še ni nobenih standardnih user-defined literalov, v C++14 pa bodo
    - "abc"s bo isto kot std::string("abc")
    - 123ms bo isto kot std::chrono::milliseconds(123) ipd. za druge časovne enote

# Unicode v string literalih

- g++ zakodira tak znak v enega ali več **char**jev po UTF-8
- Visual Studio 2013 ga zakodira v en **char** po Windows-1252; če se tistega kodnega mesta v njem ne da predstaviti, pa izpiše warning in v nizu dobimo '?'

- V string literalih se lahko sklicujemo na Unicodeova kodna mesta z \u#### ali \U######## (temu dvojemu standard pravi *universal character names*)
  - To se lahko zakodira v enega ali več znakov, odvisno od uporabljenega encodinga
  - Default encodinga standard ne predpisuje
  - Lahko pa s prefiksi zahtevamo enega od UTF encodingov: u8 za UTF-8, u za UTF-16, U za UTF-32
  - Pozor: \u zahteva natanko 4 hex števke, \U pa natanko 8
  - Za razliko od tega pa \x in \X dovolita poljubno število hex števk, vendar mora dobljeno število iti v en znak ustreznega tipa (**char**, **wchar_t**, etc.), sicer je rezultat nedefiniran

# Unicode v string literalih

- V string literalih se lahko sklicujemo na Unicodeova kodna mesta z \u#### ali \U######## (temu dvojemu standard pravi *universal character names*)
  - To se lahko zakodira v enega ali več znakov, odvisno od uporabljenega encodinga
  - Default encodinga standard ne predpisuje
  - Lahko pa s prefiksi zahtevamo enega od UTF encodingov: u8 za UTF-8, u za UTF-16, U za UTF-32
  - Pozor: \u zahteva natanko 4 hex števke, \U pa natanko 8
  - Za razliko od tega pa \x in \X dovolita poljubno število hex števk, vendar mora dobljeno število iti v en znak ustreznega tipa (**char**, **wchar_t**, etc.), sicer je rezultat nedefiniran
- Primeri:

| Literal | Tip | Vrednost (hex) |
|---|---|---|
| "a\x10d\x1f4a9" | **const char**[4] | 61, 0D*, A9*, 0 |
| L"a\x10d\x1f4a9" | **const wchar_t**[4] | 61, 10D, F4A9*, 0 |
| u8"a\u010d\U0001F4A9" | **const char**[9] | 61, C4, 8D, F0, 9F, 92, A9, 0 |
| u"a\u010d\U0001F4A9" | **const char16_t**[5] | 61, 10D, D83D, DCA9, 0 |
| U"a\u010d\U0001F4A9" | **const char32_t**[4] | 61, 10D, 1F4A9, 0 |

*Če prevajalnik kodo znaka le oklesti na spodnjih 8 ali 16 bitov. Lahko pa tudi javi opozorilo ali napako.

010D č LATIN SMALL LETTER C WITH CARON

# Unicode v string literalih

- V string literalih se lahko sklicujemo na Unicodeova kodna mesta z \u#### ali \U######## (temu dvojemu standard pravi *universal character names*)
  - To se lahko zakodira v enega ali več znakov, odvisno od uporabljenega encodinga
  - Default encodinga standard ne predpisuje
  - Lahko pa s prefiksi zahtevamo enega od UTF encodingov: u8 za UTF-8, u za UTF-16, U za UTF-32
  - Pozor: \u zahteva natanko 4 hex števke, \U pa natanko 8
  - Za razliko od tega pa \x in \X dovolita poljubno število hex števk, vendar mora dobljeno število iti v en znak ustreznega tipa (**char**, **wchar_t**, etc.), sicer je rezultat nedefiniran
- Primeri:

| Literal | Tip | Vrednost (hex) |
|---|---|---|
| "a\x10d\x1f4a9" | **const char**[4] | 61, 0D*, A9*, 0 |
| L"a\x10d\x1f4a9" | **const wchar_t**[4] | 61, 10D, F4A9*, 0 |
| u8"a\u010d\U0001F4A9" | **const char**[9] | 61, C4, 8D, F0, 9F, 92, A9, 0 |
| u"a\u010d\U0001F4A9" | **const char16_t**[5] | 61, 10D, D83D, DCA9, 0 |
| U"a\u010d\U0001F4A9" | **const char32_t**[4] | 61, 10D, 1F4A9, 0 |

UTF-8 od 010D

*Če prevajalnik kodo znaka le oklesti na spodnjih 8 ali 16 bitov. Lahko pa tudi javi opozorilo ali napako.

# Unicode v string literalih

010D č LATIN SMALL LETTER C WITH CARON
1F4A9 💩 PILE OF POO

- V string literalih se lahko sklicujemo na Unicodeova kodna mesta z \u#### ali \U######## (temu dvojemu standard pravi *universal character names*)
  - To se lahko zakodira v enega ali več znakov, odvisno od uporabljenega encodinga
  - Default encodinga standard ne predpisuje
  - Lahko pa s prefiksi zahtevamo enega od UTF encodingov: u8 za UTF-8, u za UTF-16, U za UTF-32
  - Pozor: \u zahteva natanko 4 hex števke, \U pa natanko 8
  - Za razliko od tega pa \x in \X dovolita poljubno število hex števk, vendar mora dobljeno število iti v en znak ustreznega tipa (**char**, **wchar_t**, etc.), sicer je rezultat nedefiniran
- Primeri:

| Literal | Tip | Vrednost (hex) |
|---|---|---|
| "a\x10d\x1f4a9" | **const char**[4] | 61, 0D*, A9*, 0 |
| L"a\x10d\x1f4a9" | **const wchar_t**[4] | 61, 10D, F4A9*, 0 |
| u8"a\u010d\U0001F4A9" | **const char**[9] | 61, C4, 8D, F0, 9F, 92, A9, 0 |
| u"a\u010d\U0001F4A9" | **const char16_t**[5] | 61, 10D, D83D, DCA9, 0 |
| U"a\u010d\U0001F4A9" | **const char32_t**[4] | 61, 10D, 1F4A9, 0 |

UTF-8 od 010D

UTF-8 od 1F4A9

UTF-16 od 1F4A9

*Če prevajalnik kodo znaka le oklesti na spodnjih 8 ali 16 bitov. Lahko pa tudi javi opozorilo ali napako.

# Še več novih literalov

- Raw string literals
  - Začne se z R"foo( in konča z naslednjim )foo"
  - V njem so dovoljeni celo newlinei ipd.;
    ne obdela se nobenih escape sekvenc
  - Delimiter foo je lahko kateri koli niz 0..16 znakov (ki niso ( ) \ ali whitespace)

```
const char *p = R"aaa(123")aaa456)aaa", *r = R"(a\nb
c)";
assert(strcmp(p, "123\")aaa456") == 0);
assert(strcmp(r, "a\\nb\nc") == 0);
```

- V C++14 prihajajo tudi dvojiški celoštevilski literali: 0b1001 == 9
- V C++14 lahko števke ločujemo z znakom ':

```
int a = 1234, b = 1'234, c = 1'2'3'4, d = 12'34;
printf("%d %d %d %d", a, b, c, d); // izpiše 1234 1234 1234 1234
```

# Ranged **for** statement

```
int myArr[] = { 41, 42, 43 };
for (int x : myArr) printf("%d ", x); // izpiše 41 42 43
for (char c : "ABC") printf("%d ", c); // izpiše 65 66 67 0
char *p = "ABC"; for (char c : p)  printf("%d ", c); // napaka:
                                                     // begin(char *) ne obstaja
for (int x : { 51, 52, 53 }) printf("%d ", x); // izpiše 51 52 53
vector<int> myVec = { 10, 20, 30 };
for (int &x : myVec) x += 1;   // zdaj myVec vsebuje 11, 21, 31
```

- Zdaj lahko pišemo

```
for (T myElt : mySeq) {
   …
}
```

- Kar je ekvivalentno naslednjemu:

```
{ auto&& s = mySeq;
for (auto i = begin(s), e = end(s); i != e; ++i)
{
   T myElt = *i;
   …
} }
```

  - Če obstajata s.begin() in s.end(), se kliče tadva namesto begin(s) in end(s)
  - Standardna knjižnica že vsebuje begin in end za svoje container classes, initalizer liste, arraye
  - Elemente lahko jemljemo po referenci in tako spreminjamo zaporedje: **for** (T& myElt : mySeq) { … }
  - Če tipa elementov ne poznamo / nas ne zanima, lahko uporabimo **for** (**auto** myElt : mySeq) { … } or **for** (**auto**& myElt : mySeq) { … }
  - Če hočemo dovoliti ranged **for** po svojih razredih, moramo bodisi napisati mySeq.begin() in mySeq.end() ali pa overloadati globalni begin() in end()
  - Iterator, ki ga vračata begin in end, je lahko karkoli, kar podpira !=, prefiksni ++ in dereferenciranje z *

# Function in bind

- std::bind je variadic template class, vzame poljuben callable object in sestavi nov objekt, v katerem so nekateri argumenti vezani na dane vrednosti in/ali prerazporejeni

```
int foo(int a, int b, int c);
auto bar = bind(foo, _2, 456, _1); // _1, _2, _3 itd. so iz std::placeholders
bar(100, 200); // bar bo poklical foo(200, 456, 100);
```

- std::function je wrapper okoli česarkoli, kar je mogoče poklicati

```
int foo1(int a, int b) { … }
struct S {
   static int foo2(int a, int b) { … }
   int bar(double x) {  … }
};

typedef function<int(int, int)> FooFunc;
FooFunc f1 = foo1;
FooFunc f2 = S::foo2;
FooFunc f3 = [] (int x, int y) { return x + y; };
FooFunc f4 = bind(foo, _2, 456, _1);

function<int(S*, double)> f5 = S::bar;
S s; f5(&s, 12.3); // f5 bo poklical s.bar(12.3)
function<int(double)> f6 = bind(S::bar, &s, _1);
f6(12.3); // f6 bo poklical s.bar(12.3)
```

# Generatorji naključnih števil

- Standardna knjižnica zdaj nudi več generatorjev naključnih števil z zelo natančno predpisanim obnašanjem (portable)

```
int seed = 123;
mt19937 rnd(seed); // Mersennov twister
for (int i = 0; i < 10; i++) cout << rnd(); // generira unsigned inte
```

- Obstajajo tudi razredi, ki generirajo razne verjetnostne porazdelitve

```
normal_distribution<double> dist1(12.3, 4.5); // upanje in std. dev.
for (int i = 0; i < 10; i++) cout << dist1(rnd);
uniform_int_distribution<int> dist2(5, 10); // min, max
for (int i = 0; i < 10; i++) cout << dist2(rnd);
```

  - Obstajajo *xxx*_distribution za *xxx* ∈ { uniform_int, uniform_real, bernoulli, binomial, geometric, negative_binomial, poisson, exponential, gamma, weibull, extreme_value, normal, lognormal, chi_squared, cauchy, fisher_f, student_t, discrete, piecewise_constant, piecewise_linear }

```
typedef mersenne_twister_engine<uint_fast32_t,
        32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,18,1812433253>
        mt19937;
```

3 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

# Razredi za delo s časom

- So v namespaceu std::chrono, uporabiti moramo header <chrono>
- Imamo več razredov za merjenje časa, pa tudi templatea time_point in duration s primerno podporo za aritmetiko
  - steady_clock: se zagotovljeno ves čas le povečuje
  - system_clock: to je wall clock, podpira tudi pretvorbe v/iz time_t
  - high_resolution_clock
  - Žal pa ni ničesar za merjenje porabe CPU časa našega procesa/niti

```
using namespace std::chrono;
auto t1 = steady_clock::now(); // tip t1 je steady_clock::time_point
DoSomething();
auto t2 = steady_clock::now();
auto d = t2 – t1;  // tip d je steady_clock::duration
cout << duration_cast<milliseconds>(d).count() << endl;

// Počakajmo 3 sekunde.
auto t = steady_clock::now();
while (steady_clock::now() – t < seconds(3)) { }
```

# Regularni izrazi

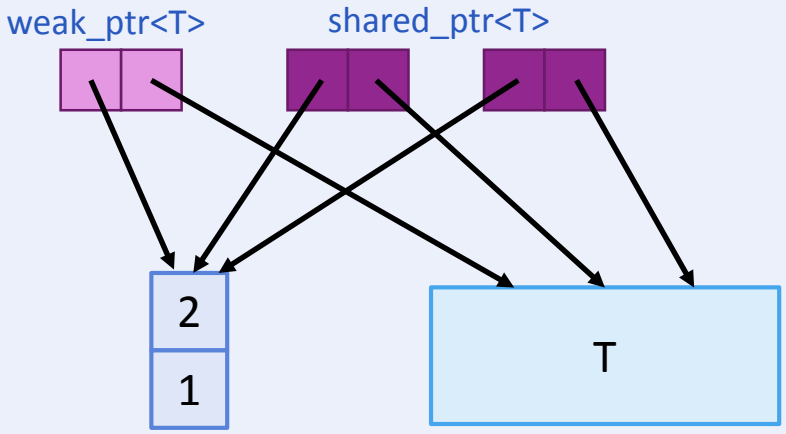
- V headerju <regex>
  - Podpirajo vse običajne stvari – grouping, character classes, backreferences
  - Iščejo lahko po **char***, po std::string in po čemerkoli, kar znamo omejiti s parom iteratorjev

```
regex re("([ab]+)(c*)");
match_results<const char*> match;
const char *text = "..abac...aaa..bbccc..ccbba..";
regex_search(text, match, re); // Najde prvo pojavitev vzorca.
cout << match.str() << " " << match[1].str() << " "
       << match[2].str() << endl; // izpiše abac aba c
for (regex_iterator<const char*> it { text, text + strlen(text), re };
      it != regex_iterator<const char*>(); ++it)
{
   const match_results<const char*> &m = *it;
   cout << m.str() << " " << m[1].str() << " " << m[2].str << endl;
}
```

  - Template parameter je tip, ki se uporablja kot iterator
    - V gornjem primeru smo uporabili **const char** *; lahko bi bil tudi std::string::const_iterator, če iščemo po std::string; lahko pa bi uporabili kakšen svoj iterator
    - Obstajajo prikladni typedefi: match_results<**const char** *> → cmatch, regex_iterator<**const char** *> → cregex_iterator itd.

# Smart pointerji

- unique_ptr ("to je edini pointer, ki kaže na ta objekt")
  - Njegov destruktor pobriše objekt, na katerega kaže kazalec
    - Kaj točno pomeni "pobriše", se da customizirati
  - Ni ga mogoče kopirati, lahko pa ga premikamo
  - Podoben staremu auto_ptr [ki je zdaj deprecated], le da je po zaslugi move semantics v C++11 narejen bolje
- shared_ptr, weak_ptr
  - Poleg kazalca na objekt imata tudi kazalec na par števcev ⟨use count, weak use count⟩
    - Use count = število shared_ptrjev, ki kažejo na ta objekt
    - Weak use count = število weak_ptrjev, ki kažejo na ta objekt
  - Ko use count pade na 0, se objekt pobriše
  - Ko oba use counta padeta na 0, se pobriše tudi par števcev
  - shared_ptr = "mi smo (eden od) lastnik(ov) tega objekta"
  - weak_ptr = "mi smo le uporabnik tega objekta, če ta sploh še obstaja"
    - weak_ptr nam sploh ne dovoli do pointerja na objekt, lahko pa nam ustvari nov shared_ptr nanj



# Novi containerji v standardni knjižnici

- Hash tabele:
  - unordered_map, unordered_set,
  - unordered_multimap, unordered_multiset
- array<T, N> je wrapper okrog arraya N elementov tipa T,
  ki se obnaša kot STLovski container
  - Definira iteratorje, .begin(), .end(), .at(), .front(), .back() ipd.
- Singly-linked list: forward_list
- Emplacement:
  - Mnogi containerji zdaj podpirajo operacijo, ki skonstruira nov element v containerju,
    tako da ni treba niti kopiranja niti premikanja

```
struct S {
    S(int x, char *y) { … }
};
vector<S> v;
v.emplace_back(123, "foo"); // to je, mimogrede, variadic template member funkcija
```

# Niti

- Novo nit zaženemo tako, da ustvarimo instanco std::thread in ji damo nekaj, kar lahko pokliče

```
void foo1(int a, char *b) { ... }
struct S2 { void foo2(int a, char *b) { ... } };
struct S3 { void operator()(int a, char *b) { ... } };

thread t1 { foo1, 123, "blah blah" };
S2 s2; thread t2 { &S2::foo2, &s2, 123, "blah blah" };
S3 s3; thread t3 { s3, 123, "blah blah" };
    // opomba: t3 naredi svojo kopijo s3 in na njej pokliče operator()
thread t4; // jo bomo zagnali kasneje
this_thread::sleep_for(seconds(3));
t4 = thread([] () { printf("Hello world"); }); // poženimo jo zdaj
t1.join(); t2.join(); t3.join(); t4.join(); // počakajmo, da končajo
```

# Thread-local storage

- S ključno besedo **thread_local** dosežemo, da ima vsaka nit svojo kopijo spremenljivke
  - Destruktor se kliče, ko se nit konča

```
struct S {
    S() { printf("%p created\n", this); }
    ~S() { printf("%p destroyed\n", this); }
};
struct OtherClass {
    static thread_local S s;
};
thread_local S s; // globalna spremenljivka
thread_local S OtherClass::s; // static class member
S& GetS() { // tudi za statične lokalne spremenljivke
    static thread_local S s; return s; }
```

- Podpora tem stvarem še malo šepa
  - Visual Studio 2013 besede **thread_local** sploh ne podpira
  - Pri g++ sem opazil, da se destruktorji **thread_local**nih objektov kličejo le, če so to statične lokalne spremenljivke,
    ne pa pri globalnih spremenljivkah in static class memberjih

# Muteksi

- Muteksi so zdaj del standardne knjižnice:

```
mutex m;
int x; // podatki, ki si jih deli več niti

void MyThreadFunc() {
    …
    m.lock();
    … // naredi nekaj z x
    m.unlock();
    …
}
```

- Podpira tudi m.try_lock()
- Za ljudi, ki so na mutekse navajeni iz Windowsov: C++ov mutex je precej cenejši in preprostejši sinhronizacijski objekt kot mutex iz Windows APIja
  - V Visual Studiovi implementaciji standardne knjižnice je muteks implementiran z InterlockedExchangePointer
- Sorodni razredi so še:
  - timed_mutex: podpira .try_lock_for(timeout)
  - recursive_mutex: ga lahko ista nit zaklene večkrat hkrati
  - recursive_timed_mutex
  - shared_mutex v C++14: poleg lock (= exclusive) bo podpiral še .lock_shared(), .unlock_shared()

# Locks

- Kaj če pozabimo odkleniti muteks?
- Uporabimo raje ključavnico:

```
void MyThreadFunc() {
    ...
    {
        lock_guard<mutex> lck(m); // konstruktor pokliče m.lock()
        ... // naredimo nekaj z x
    } // lck pade iz scopea, njen destruktor kliče m.unlock()
    ...
}
```

- Vidimo, da je to template – namesto mutexa mu lahko damo karkoli, kar ima .lock() in .unlock()
- Obstaja tudi unique_lock<M>, ki je kot lock_guard z dodatnimi operacijami
  - .lock(), .unlock(), .owns_lock() → **bool**, try_lock(), try_lock_for(timeout)

# Condition variables

- Ena ali več niti lahko čaka na pogojno spremenljivko, druga nit jo/jih lahko zbudi

```
queue<int> q; // skupni podatki
mutex mx;
condition_variable cv;

void Producer()
{
   …
   { unique_lock<mutex> lck(mx);
   q.push(x); }
   cv.notify_all();
   …
}
```

```
void Consumer()
{
   while (! stopFlag) {
      unique_lock<mutex> lck(mx); // zaklene mx
      cv.wait(lck); // wait odklene lck, počaka in jo spet zaklene
      while (! q.empty()) {
         int x = q.front(); q.pop(); DoSomethingWith(x); }
   } // lck v destruktorju odklene mx
}
```

# Atomarni tipi

- atomic<T> hrani vrednost tipa T in omogoča atomaren dostop do nje
  - .load(), .store(), .exchange(), compare_exchange
  - Za T = int in podobne tipe podpira tudi ++, --, +=, –=, &=, |=, ^=
  - Deluje za poljuben tip T, vendar:
    - Splošna implementacija uporablja ključavnico, da sinhronizira dostope do vrednosti
    - Za majhne tipe T obstajajo specializirane implementacije, ki uporabljajo cenejše mehanizme brez zaklepanja (interlocked increment itd.)

# Task-based concurrency

- Asinhronski klic funkcije (oz. česarkoli):

```
double MySlowFunc(int a, int b) {
    this_thread::wait_for(seconds(3)); return a / double(b); }

auto fu = async(MySlowFunc, 10, 20); // fu je tipa std::future<double>
...
double x = fu.get();
printf("%d", x); // izpiše 30
```

  - .get() počaka, da se asinhronsko opravilo konča
  - async lahko požene novo nit, lahko ima nek pool z nitmi,
    lahko pa se celo odloči, da bo klic izvedla sinhronsko (šele takrat, ko mi pokličemo .get())
  - Lahko pa mu povemo, kaj od tega naj naredi (*launch policy*)

```
auto fu = async(launch::async, MySlowFunc, 10, 20);
while (fu.wait_for(milliseconds(300) != future_status::ready)
    printf("Still waiting...\n");
double x = fu.get();
```

# Task-based concurrency

- Nekatere stvari, ki jih uporablja async, so nam na voljo tudi neposredno
  - promise<T> je struktura, kamor asinhronsko opravilo odloži svoj rezultat (tipa T), do katerega se bo potem dalo dostopati prek future<T>

```
promise<double> pr;
future<double> fu = pr.get_future();
thread thr { MyWrapper, pr, 10, 20 };
```

```
void MyWrapper(promise<double>& pr, int a, int b)
{
    double result = MySlowFunc(a, b);
    pr.set_value(result);
}
```

```
double x = fu.get();
```

  - Pravzaprav takšen wrapper že obstaja – imenuje se packaged_task
    - Zna tudi poloviti izjeme, ki jih sproži MySlowFunc (in ki jih lahko potem vidimo skozi future)

```
packaged_task<double(int, int)> pt { MySlowFunc };
future<double> fu = pr.get_future();
```

```
pt(10, 20); // klicano v neki drugi niti
```

```
double x = fu.get();
```

# Več informacij

- ISO C++ Standards Committee http://www.open-std.org/jtc1/sc22/wg21/
  - C++11 draft http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
  - C++14 draft http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf
- Stroustrup: The C++ Programming Language, 4th edition http://www.stroustrup.com/4th.html
- Stroustrupov C++11 FAQ http://www.stroustrup.com/C++11FAQ.html
- Wikipedija http://en.wikipedia.org/wiki/C++11 http://en.wikipedia.org/wiki/C++14
- Rvalue reference in z njimi povezane stvari:
  - Alex Allain http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
  - Thomas Becker http://thbecker.net/articles/rvalue_references/section_01.html
  - Scott Meyers http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers