Променљиве и функције

Име

- Главне ствари које именујемо у Це++-у су:
  - променљиве
  - функције
  - типови

- Придруживање имена ономе што треба да означава, дешава се приликом декларације.

Декларација

- Декларација описује језички појам (променљиву, функцију, тип...), дајући довољно информација да компајлер може знати како са тим појмом треба да поступа, тј. како да га **користи**.
- Декларација саопштава две кључне ствари:
  - уводи назив, тј. име, тог појма (у досег)
  - одређује тип тог појма (до неке мере, обично у потпуности)
- У Це++-у појам мора бити декларисан пре него што се може користити (сем једног изузетка)
- Примери:
  - **int a = 7;**
  - **const double cd = 8.7;**
  - **double sqrt(double x);**
  - **vector<Token> v;**
  - **class My_class;**

Дефиниција

- Дефиниција у потпуности описује неки појам. Потпуне информације су потребне да би компајлер знао како да **направи** тај појам.
- Код променљивих и функција, прављење подразумева заузимање ресурса за тај појам, као и његову иницијализацију.
- Код променљивих (скоро) све декларације су уједно и дефиниције. Постоје само два изузетка, на које ћемо се осврнути ако буде било времена и потребе.
  - Примери дефиниција:

```
int a = 7;
int b;
vector<double> v;
double sqrt(double x) { … };
struct Point { int x; int y; };
```

  - Пример декларација, без дефиниције:

```
double sqrt(double x); // недостаје тело функције
// али оно није ни потребно да би се функција позвала
```

Декларација и дефиниција

- Не могу бити две дефиниције истог појма
  - На пример:

```
int a;
int a;
double sqrt(double d) { … }
double sqrt(double d) { … }
```

- Могу постојати две декларације истог појма. На пример:

```
double sqrt(double d); // декларација
double sqrt(double d) {
  …
} // дефиниција (уједно и декларација)
```

Декларација и дефиниција

- Зашто постоји и једно и друго?
- Да бисмо нешто користили, довољна нам је само декларација.
- Најчешће се дефиниција налази негде другде
  - Касније у истој датотеци
  - Или у сасвим другој датотеци – написана од стране сасвим друге особе
- Декларације одређују **спрегу**
  - ка остатку кода
  - ка библиотекама
    - библиотеке су важне: не можемо све сами написати, а и зашто би?
- Савет:
  - Декларације иду у заглавље да би се лако користиле, тј. убациле у друге датотеке

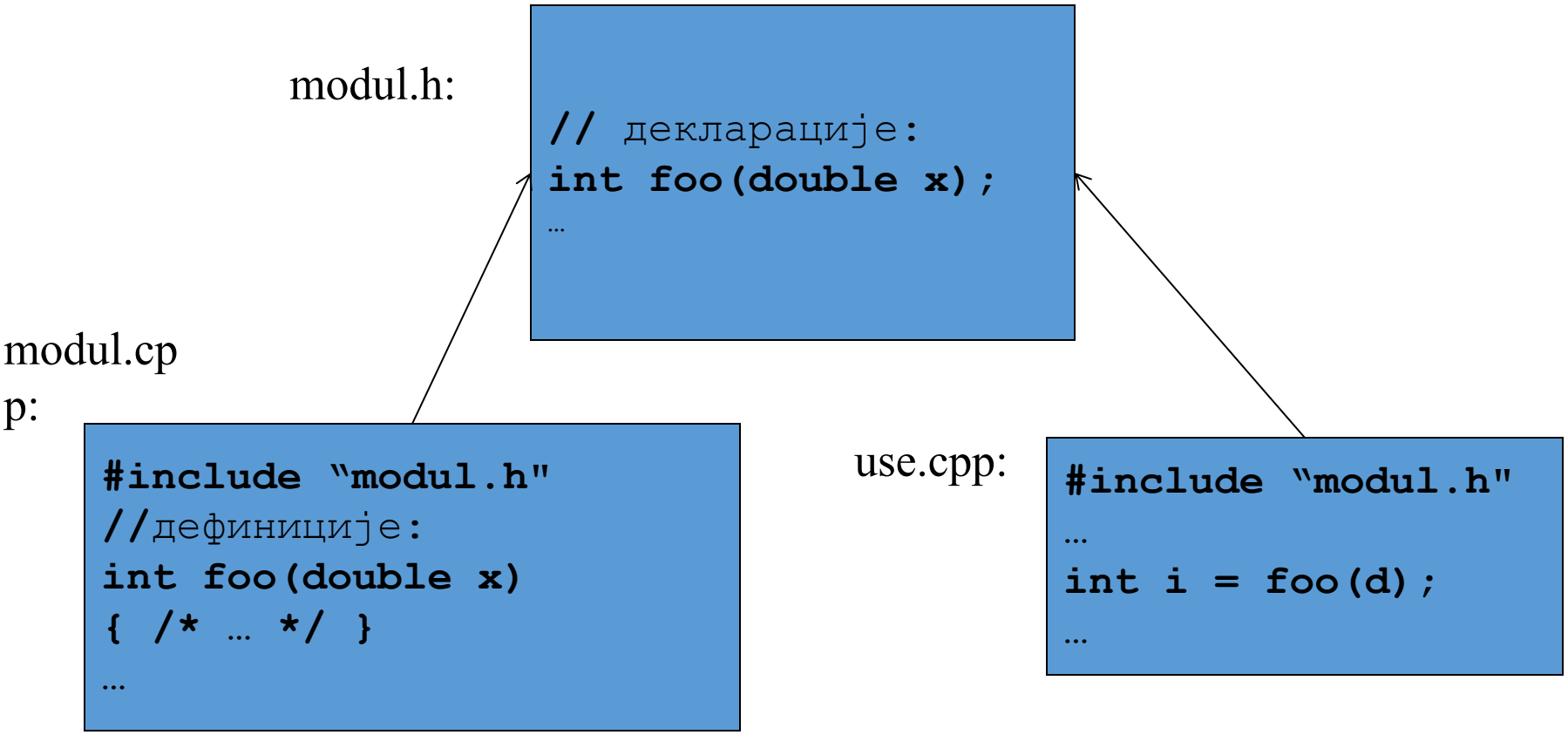
Заглавље и претпроцесор

- Заглавље је датотека која садржи декларације функција, типова, променљивих; и осталог.
- Конструкција:

```
#include "std_lib_facilities.h"
```

  је (претпроцесорска) директива која додаје те декларације у код.
- Заглавље омогућава приступ функцијама, типовима итд. које желите да користите у свом програму.
  - Обично вас не занима како су те ствари заправо имплементиране.

Изворни код



- Иста **#include** конструкција у обе **.cpp** датотеке олакшава одржавање конзистентности

Досег

- Досег је део програмског кода:
  - Рецимо:
    - Датотечки досег, тј. глобални досег (ван било које друге језичке конструкције)
    - Блоковски досег, тј. локални досег (у блоку {...}, или у листи параметара код дефиниције функције)
    - Класни досег (унутар дефиниције класе)
    - Исказни досег (унутар једног исказа, нпр. for исказа:

      for(int i = 0; i < 50; ++i)
- Име које уведено у неком досегу видљиво је (може му се обратити) из тог истог досега (обично након тачке увођења – класе су изузетак), и из свих досега који су у њему угнијеждени.
- Досези групишу и **ограничавају** идентификаторе на део кода којег се тичу.

Досег

```
void foo(int x) {
   ...
   float y;
   ...
}


void bar(double x) {
   ...
   string y;
   ...
}
```

Досег

```
#include "std_lib_facilities.h" // max и abs су декларисани овде
// r, i и v нису видљиви одавде
class My_vector {
 vector<int> v; // v је класног досега
public:
 int largest() // largest је класног досега
 {
     int r = 0; // r је локална, тј. блоковског досега
     for (int i = 0; i < v.size(); ++i) // i је исказног досега
          r = max(r,abs(v[i]));
     // i није видљиво из ове тачке
     return r;
 }
 // r није видљиво (v је видљиво)
};
// v није видљиво
```

Досег

```
int x; // глобалне променљиве
int y; // обично их не треба користити у оваквом чистом облику

int f()
{
  int x; // локална променљива – сада постоје два x-а
  x = 7; // односи се на локални x, не на глобални
  {
    int x = y; // још једана локална променљива x
               // (сада имамо три x)
    x++;       // односи се на локално x у овом досегу
  }
}

// овакве кодове не треба (свесно) писати!
```

Функције

Дефиниција

```
return-type function-name(parameters)
{
  declarations
  statements
  return value;
}
```

- **return-type** - тип повратне вредности; void ако нема повратне вредности
- **function-name** - јединствено име функције (функција и променљива истог досега не могу се звати исто)
- **parameters** - листа декларација променљивих које представљају параметре функције, међусобно су одвојене зарезом
- **return value;** - вредност која ће бити враћена након завршетка функције (није неопходно ако је тип повратне вредности `void`)

Декларација

```
return-type function-name(parameter-types);
```

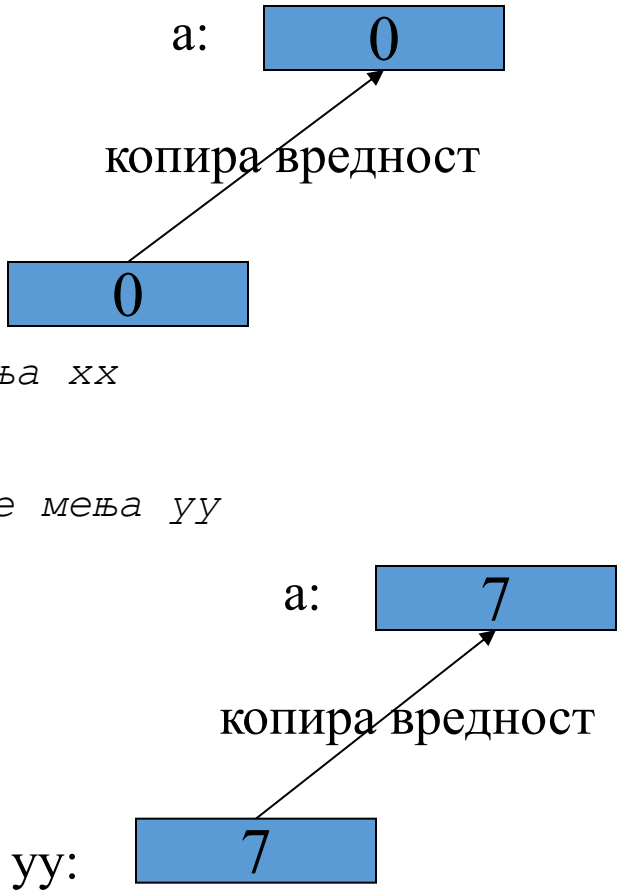
- **parameter-types** - листа типова параметара; за разлику од листе у дефиницији, овде се не морају навести имена параметара (али је то ипак у пракси пожељно!)

```
int foo(float, int, const long*);
```

Прослеђивање параметара по вредности

```
// шаље функцији копију тренутне вредности стварног параметра
int f(int a) { a = a + 1; return a; }

int main()
{
  int xx = 0;
  cout << f(xx) << endl; // исписује 1
  cout << xx << endl; // исписује 0; f() не мења xx
  int yy = 7;
  cout << f(yy) << endl; // исписује 8;  f() не мења yy
  cout << yy << endl; // исписује 7
}
```

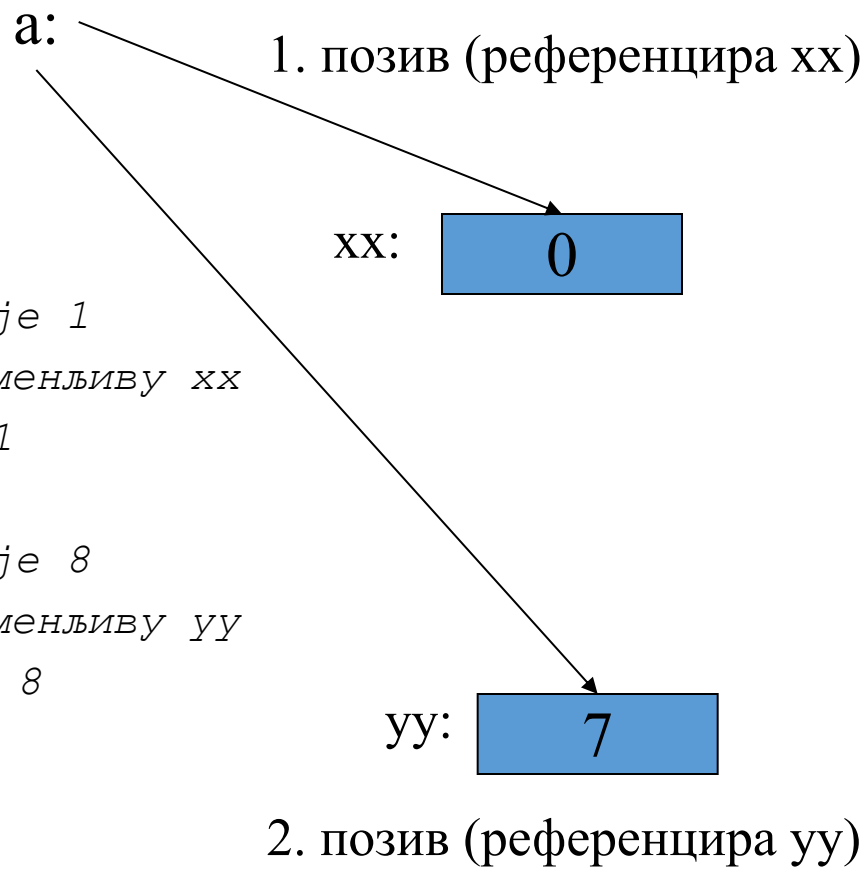


Прослеђивање параметара по референци

```
// шаље функцији референцу на стварни параметар
int f(int& a) { a = a + 1; return a; }


int main()
{
  int xx = 0;
  cout << f(xx) << endl; // исписује 1
            // f() је изменила променљиву xx
  cout << xx << endl; // исписује 1
  int yy = 7;
  cout << f(yy) << endl; // исписује 8
            // f() је изменила променљиву yy
  cout << yy << endl;  // исписује 8

}
```



Поређење два механизма прослеђивања параметара

По вредности:
- Вредност стварног параметра се копира
- Стварни параметар се не може изменити

По референци:
- Нема копирања
- Стварни параметар се може изменити

Поређење два механизма прослеђивања параметара

По вредности:
- Вредност стварног параметра се копира
- Стварни параметар се не може изменити

По референци:
- Нема копирања
- Стварни параметар се може изменити

- **Копирање није (увек) бесплатно!**
- **Безбедност је важна (ако функција неће мењати параметар, не треба јој то ни дозволити)**

Поређење два механизма прослеђивања параметара

По вредности:
- Вредност стварног параметра се копира
- Стварни параметар се не може изменити

По референци:
- Нема копирања
- Стварни параметар се може изменити

- **Копирање није (увек) бесплатно!**
- **Безбедност је важна (ако функција неће мењати параметар, не треба јој то ни дозволити)**

По константној референци:
- Нема копирања
- Стварни параметар се не може изменити

```
int f(const int& a) { a = a + 1; return a; }
int f(const int& a) { int t = a + 1; return t; }
int f(const int& a) { return a + 1; }
```

Кад користити који механизам - савети

- По вредности – када су параметри само улази и када променљиве нису велике. Уједно, приступ променљивој је обично (мало) бржи када је директан, а не преко референце.
- По константној референци – када су параметри само улази и када су променљиве велике
- По референци:
  - Када је потребно више од једног излаза из функције, нпр.:

```
void splitVector(const vector<int>& x,
                 vector<int>& y,
                 vector<int>& z);
```

  - Када је потребно изменити неку већу променљиву

```
vector<int> add0ToEnd(vector<int> x) {
    x.push_back(0);
    return x;
}
void add0ToEnd(vector<int>& x) {
    x.push_back(0);
}
```

Илустрације

```
void crfoo(const int& x);
void rfoo(int& x);
void foo(int x);

int a;
const int c = 5;
```

```
foo(a);   // OK      foo(c);   // OK
rfoo(a);  // OK      rfoo(c);  // Error!
crfoo(a); // OK      crfoo(c); // OK

foo(5);   // OK
rfoo(5);  // !!
crfoo(5); // OK
```

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

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

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

```
bar(a); -> bar(int&)

bar(c); -> bar(int const&)
```

Резултујући асемблерски код је идентичан:

```
movl    $a, %edi
call    bar(int&)
movl    $c, %edi
call    bar(int const&)
```

Шта одређује функцију? - подсећање

- Функција **није** једнозначно одређена само својим именом.
- Једнозначно је одређена именом, бројем параметара и њиховим типовима. То се зове још и „потпис функције“.
- Тип повратне вредности не одређује функцију.

```
double sqrt(float x);
float sqrt(float x); // ово не може
```

- Такође, овај пример илуструје да је const део потписа:

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

bar(a); -> bar(int&)
bar(c); -> bar(int const&)
```

Референце

- Концепт референци није везан само за прослеђивање параметара.
- Али за сада само оволико :)

Именски простори

- Размотримо код двоје програмера: Сретена и Добриле

```
class Glob { /*…*/ };   // у Сретеновом заглављу sreten.h
class Widget { /*…*/ }; // исто у sreten.h

class Blob { /*…*/ };   // У Добрилином заглављу dobrila.h
class Widget { /*…*/ }; // исто у dobrila.h

// наш код:
#include “sreten.h"
#include “dobrila.h"

void my_func(Widget p)  // грешка: који Widget?
{
  // …
}
```

Именски простори

- Именски простори су један начин да се ови проблеми превазиђу (или бар умање):

```
namespace Sreten { // у Сретеновом заглављу
    class Glob{ /*…*/ };
    class Widget{ /*…*/ };
}

// наш код:
#include “sreten.h"
#include “dobrila.h"

void my_func(Sreten::Widget p) // сад је све јасно
{
    // …
}
```

Именски простори

- Именски простор је заправо глобални досег са називом
- Операција :: се користи да би се саопштило у ком досегу се налази појам који желимо да употребимо
- На пример, **cout** припада именском простору **std**, и да би му се обратили у суштини морамо писати:

```
std::cout << "…";
```

- Један именски простор може обухватати више блокова у више различитих датотека.

```
namespace Sreten { // у заглављу sreten1.h
    class Glob{ /*…*/ };
}
namespace Sreten { // у заглављу sreten2.h
    class Widget{ /*…*/ };
}
```

Именски простори

- Али то до сада нисмо писали. Како то?
- Стално писање именског простора неког појма може бити напорно и често оптерећује код.
  - Зато можемо декларисати коришћење појма из неког именског простора („using declaration“)

```
using std::cout; // када напишем cout мислим на std::cout
cout << "…"; // ово је у ствари std:cout
cin >> x; // грешка: cin не постоји у подразумеваном досегу
```

  - или декларисати коришћење целог именског простора („using directive“)

```
using namespace std; // све из std подижемо у датотечки досег
cout << "…"; // мисли се на std::cout
cin >> x; // мисли се на std::cin
```

- У нашем заглављу std_lib_facilities.h налазе се одговарајуће јузинг декларације.

Функције и методе

Ментални модел који може помоћи код разумевања како се заправо функција чланица разликује од слободне функције.

```
class MyClass {
public:
  int m;
  void func(int n) {
    m = n;
  }
};

void func(MyClass& a, int n) {
  a.m = n;
}
```

```
MyClass x;

x.func(5);

func(x, 5);
```

Резултујући асемблерски код:

```
movl $5, %esi
movl $x, %edi
call MyClass::func(int)

movl $5, %esi
movl $x, %edi
call func(MyClass&, int)
```

Функције и методе

Раздвајање декларације и дефиниције код функција чланица (метода).

```
class MyClass {
  int m;
public:
  void func(int n);
};

void MyClass::func(int n) {
  m = n;
}
```

Илустрација спреге ка класама везаним за токене