

# Algoritmi i strukture podataka

- predavanja -

---

## 2. C++

# Osnovno o jeziku

- Autor Bjarne Stroustrup, 1985.
- proširenje jezika C objektnom paradigmom
- "C with Classes"
- imperativni i proceduralni jezik
  - stanje programa mijenja se naredbama koje su sadržane u procedurama
  - to je različito od deklarativnog jezika, gdje se opisuju svojstva rezultata – to nazivamo funkcijskom, logičkom ili matematičkom paradigmom
- sve što se može u C-u, može se i u C++, ali i
  - objektno orijentiran
  - ima mogućnost generičkog programiranja (nije potrebno pisati nove funkcije (iste funkcionalnosti) za druge tipove podataka)

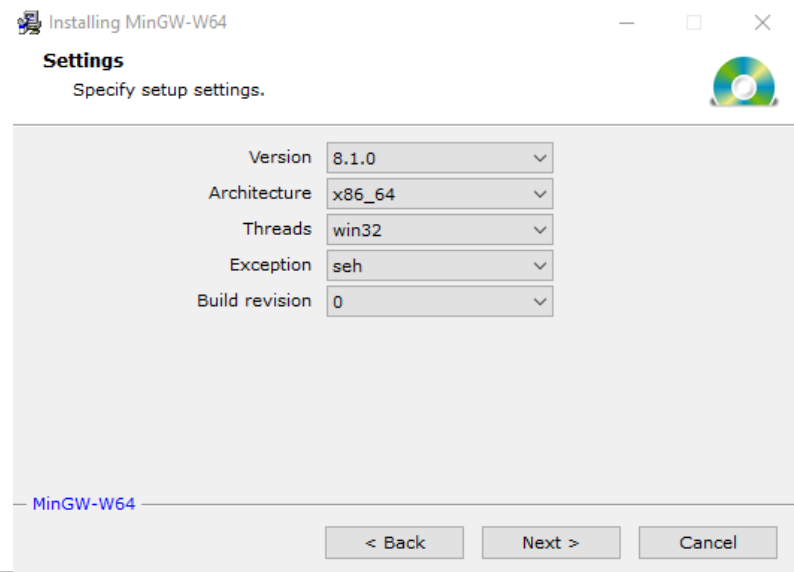
# Hello, world!

- Izvorna datoteka (Linux obično .cc, Windows obično .cpp)

```
#include <iostream>
int main(void) {
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

HelloWorld.cpp

- Instalirati <https://sourceforge.net/projects/mingw-w64/>  
u neku mapu koja u nazivu  
nema praznine (npr.  
C:\mingw)  
sa sljedećim postavkama:



# Hello, world!

- Postaviti PATH na ...\\bin
- Prevođenje

```
g++ -g -o HelloWorld.exe HelloWorld.cpp  
ili  
gcc -g -o HelloWorld.exe HelloWorld.cpp -lstdc++
```

- Izlaz

```
Hello world!↵
```

# Prostor imena (*namespace*)

- Omogućava postojanje više programskih elemenata istog imena
- Operator određivanja pripadnosti `::` (*scope resolution operator*)

```
#include <iostream>
namespace s1 {
    float const pi = 3.14;
}
namespace s2 {
    double const pi = 3.141592654;
}
int main(void) {
    std::cout << s1::pi << " ";
    std::cout << s2::pi << std::endl;
    return 0;
}
```

Namespace.cpp

3.14 3.14159 ↵

# Prostor imena (*namespace*)

- Naredba `using` omogućava korištenje programskih elemenata bez `::`:

```
#include <iostream>
namespace s1 {
    float const pi = 3.14;
}
namespace s2 {
    double const pi = 3.141592654;
}
using namespace s1;
int main(void) {
    std::cout << pi << std::endl;
    std::cout << s2::pi << std::endl;
    return 0;
}
```

NamespaceUsing.cpp

3.14 3.14159↵

# Prostor imena (*namespace*)

- Naredba `using namespace` određuje pretpostavljene prostore u kojima prevodilac traži identifikatore
- *Hello, world* s `using namespace`

```
#include <iostream>
using namespace std;
int main(void) {
    cout << "Hello world!" << endl;
    return 0;
}
```

HelloWorld.cpp

Hello world! ↵

- vrijedi od mjesta navođena naredbe pa do kraja bloka u kojem je navedena
- ako je navedena izvan bloka, vrijedi do kraja datoteke
- može se navesti više `using` naredbi, ali treba paziti da ne dođe do neodređenosti, ako se neki simbol s istim nazivom nalazi u više od jednog prostora imena

# Jednostavni ulaz/izlaz

- Standardne funkcije programskog jezika C (printf, scanf itd.)
- Globalni objekti cout, cin, cerr s operatorima << i >>

```
...  
    double v;  
    cin >> v;  
    if (v < 0) cerr << "Negativni broj";  
    else cout << sqrt(v);  
...
```



# Višeznačnost funkcija (*function overloading*)

- Funkcije se prepoznaju prema potpisu (*signature*):
  - naziv funkcije, broj i tipovi parametara

Overloading.cpp

```
...  
int kvadrat(int arg) {  
    cout << "int ";    return arg * arg;  
}  
double kvadrat(double arg) {  
    cout << "double ";    return arg * arg;  
}  
int main(void) {  
    cout << kvadrat(2) << " " << kvadrat(2.) << " "  
        << kvadrat('A') << endl;  
    return 0;  
}
```

```
int 4 double 4 int 4225↵
```

# Višeznačnost funkcija (*function overloading*)

- Ako se funkcija pozove s nekim drugim tipom argumenta, obaviti će se automatska konverzija, ako je moguće, npr.

```
...  
int kvadrat(int arg) {  
    cout << "int ";    return arg * arg;  
}  
double kvadrat(double arg) {  
    cout << "double ";    return arg * arg;  
}  
int main(void) {  
    cout << kvadrat(2.f) << endl;  
    return 0;  
}
```

double 4.0

# Referenca (&)

- Nepromjenjivi pokazivač kojeg prevodilac automatski dereferencira

```
#include <iostream>
using namespace std;

int main(void) {
    int a = 1;
    int &b = a;
    // slično kao int *b = &a;
    // ali bi kasnije trebalo eksplicitno dereferencirati
    b = 7;
    cout << a << " " << b;
    return 0;
}
```

Reference.cpp

# Referenca (&)

- Najčešće se koristi kod parametara funkcije

```
#include <iostream>
using namespace std;
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main(void) {
    int a = 1, b = 2;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}
```

ReferenceFunction.cpp

# Razred (*class*)

- Skup istovrsnih objekata koji sadrže podatke (članske varijable, *members*) i nad kojima su definirane funkcije (*methods*, *member functions*)
- Deklaracija razreda

Point.cpp

```
...
class Point {
public:
    double x;
    double y;
    void mirrorX(double xValue) { x = xValue - (x - xValue); }
    void mirrorY(double yValue) { y = yValue - (y - yValue); }
};
...
```

- Kao i kod strukture u C-u, ovo je samo predložak (nacrt, *blueprint*) budućih objekata

# Razred (*class*)

- `public` izlaže članske varijable i funkcije prema ostatku programa
- ako se ne navede, članske varijable i funkcije su `private` i ne mogu se koristiti izvan razreda
- detaljnije o `public`, `private` i `protected` kasnije

# Razred (*class*)

- Stvarni objekt treba stvoriti (instancirati) definicijom, čime se rezervira memorija, ali ne inicijaliziraju članske varijable

```
int main(void) {  
    Point p;  
    cout << sizeof(p) << endl;  
    cout << p.x << " " << p.y << endl;  
    p.x = 10;  
    p.y = 20;  
    p.mirrorX(-1);  
    cout << p.x << " " << p.y << endl;  
    return 0;  
}
```

```
16  
7.00584e-317 7.90505e-323  
-12 20
```

# Konvencije za nazivlje

- U C++ standardu svi nazivi započinju malim slovom
- Ali bolje je i široko prihvaćeno:
  - Nazive razreda započinjati velikim slovom kako bi se razlikovali od konkretnih objekata  
`class Point { ...`
  - Funkcije i varijable započinjati malim slovom, koristiti "*Camel casing*"  
`x, mirrorX, xAxis`
  - konstante pisati velikim slovom  
`const double PI=3.141592654;`
- Ponekad se preporuča privatne varijable započinjati s `m_`, a parametre funkcija s `t_`
- Novije konvencije predlažu `_` na kraju ili, još bolje, na početku imena privatne varijable



# Razred (*class*)

- Opis složenijeg razreda uobičajeno se rastavlja u dvije datoteke
- Datoteka zaglavlja, s deklaracijom razreda, npr. Point.h

```
class Point {  
    public:  
        double x;  
        double y;  
        void mirrorX(double xValue);  
        void mirrorY(double yValue);  
};
```

- Datoteka s definicijom članskih funkcija, koja treba #include, ali i obvezno ::, npr. Point.cpp

```
#include "Point.h"  
void Point::mirrorX(double xValue) { x = xValue - (x - xValue); }  
void Point::mirrorY(double yValue) { y = yValue - (y - yValue); }
```

# Razred (*class*)

- Glavni program treba `#include`, npr. `Main.cpp`

```
#include <iostream>
#include "Point.h"
using namespace std;
int main(void) {
    Point p;
    ...
    return 0;
}
```

- Prevođenje i povezivanje

```
g++ -g -o Main.exe Point.cpp Main.cpp
```

- Zbog jednostavnosti i lakšeg prevođenja i povezivanja, većina će primjera biti napisana u jednoj datoteci koja objedinjuje deklaracije, definicije i glavni program

# Razred (*class*)

- Inicijalne vrijednosti članskih varijabli i operator pridruživanja

```
#include <string.h>
```

PointAssign.cpp

```
...
```

```
int main(void) {  
    Point p, r, s;  
    cout << p.x << " " << p.y << endl;  
    p.x = 10; p.y = 20;  
    r = p;  
    cout << r.x << " " << r.y << endl;  
    memcpy(&s, &p, sizeof(p));  
    cout << s.x << " " << s.y << endl;  
    return 0;  
}
```

```
1.79158e-307 1.79173e-307↵
```

```
10 20↵
```

```
10 20↵
```

# Razred (*class*)

- Pristup članovima preko pokazivača

```
...  
int main(void) {  
    Point p, *r;  
    r = &p;  
    cout << r->x << " " << r->y << endl;  
    r->mirrorX(); // ili (*r).mirrorX();  
    cout << (*r).x << " " << (*r).y << endl;  
}
```

# Konstruktori

- Funkcije koje se obavljaju prilikom definicije, najčešće služe za inicijalizaciju vrijednosti članskih varijabli

PointConstructor.cpp

```
#include <iostream>
using namespace std;
class Point {
public:
    double x;
    double y;
    Point() { // pretpostavljeni (default) konstruktor
        x = 0; y = 0;
    }
    Point(double xIn, double yIn) { // konstruktor s parametrima
        x = xIn; y = yIn;
    }
};
```

# Konstruktori

```
...  
int main(void) {  
    Point p, r(1, 2);  
    cout << p.x << " " << p.y << endl;  
    cout << r.x << " " << r.y << endl;  
    return 0;  
}
```

0 0 ↵

1 2 ↵

# Konstruktori s inicijalizacijskim listama

- Ekvivalent prethodnom primjeru

```
...
class Point {
    public:
        double x;
        double y;
        Point() : x(0), y(0) {}
        Point(double xIn, double yIn) : x(xIn), y(yIn) {}
};
...
```

**PointConstructorInitList.cpp**

# Konstruktori s inicijalizacijskim listama

- Ovako se parametri mogu zvati isto kao i članske varijable

```
...  
class Point {  
    public:  
        double x;  
        double y;  
        Point() : x(0), y(0) {}  
        Point(double x, double y) : x(x), y(y) {}  
};  
...
```



# Konstruktori

- Inače, problem:

```
...  
class Point {  
    public:  
        double x;  
        double y;  
        Point(double x, double y) {  
            x = x;  
            y = y;  
        }  
};  
...  
Point r(1, 2);  
cout << r.x << " " << r.y << endl;
```

-8.31488e+150 1.34903e+264 ↵

# Pokazivač `this`

- Sadrži adresu aktivnog objekta

```
...  
class Point {  
    public:  
        double x;  
        double y;  
        Point(double x, double y) {  
            this->x = x;  
            this->y = y;  
        }  
};  
...  
Point r(1, 2);  
cout << r.x << " " << r.y << endl;
```

1 2 ↵

# Destruktor

- Funkcija koja se poziva u trenutku uništavanja objekta
- Oslobađa resurse koje je zauzeo objekt  
`~SomeClass() { ... }`
- Primjer: primitivna realizacija razreda `File` u kojemu se datoteka otvara u konstruktoru, a zatvara u destrukturu

**Destructor.cpp**

# Uniformna inicijalizacija {}

- Jedinstven način inicijalizacije varijabli, objekata i polja
- Omogućava i stvaranje novih objekata pri pozivu funkcije ili povratku iz funkcije

**UniformInitialization.cpp**

# Nasljeđivanje

- Primjer: vrlo pojednostavnjeni skup geometrijskih likova
  - trokut, kvadrat, pravokutnik, krug zadani su identifikatorom, koordinatom referentne točke (donji lijevi ugao za trokut, kvadrat, pravokutnik, središte za krug), osnovicom  $a$  (uvijek paralelna s apscisom) ili radijusom, te drugom ( $b$ , pravokutnik) i trećom stranicom ( $c$ , trokut)
  - Treba implementirati skup likova i omogućiti izračun ukupne površine svih likova.

# Put prema nasljeđivanju

- Vrlo primitivno rješenje

**ShapeElementary.cpp**

- Ali kako organizirati skup po kojem se može iterirati?
- Prljavo i grozno rješenje, ali jedino moguće bez nasljeđivanja
  - `void *set[size];`
  - Kod dereferenciranja treba *cast* u neki konkretni lik, ali u koji?

**ShapeDirty.cpp**

- Dodatni problemi:
  - svaki lik definira i zajedničke elemente koji se u deklaraciji moraju svaki puta pobrojati (`id`, `x`, `y`, `a`)
  - `id` se može promijeniti bilo gdje u kôdu
- Pravo rješenje: nasljeđivanje kojim se ostvaruje višeobličje (polimorfizam) **ShapeInheritance.cpp**

# Nasljeđivanje funkcija i konstruktora

- Ako u nekom razredu ne postoji funkcija sa određenim potpisom, potražit će se takva funkcija u nadređenom razredu
- U slučaju konstruktora, sve do C++11 trebalo je eksplicitno pozivati konstruktor iz nadređenog razreda iako konstruktor nadređenog razreda ima iste parametre

```
class Square {  
    ...  
    Square(int id, double x, double y, double a) : Shape(id, x, y, a) {}  
    ...  
}
```

- Od C++11 dovoljno je koristiti ključnu riječ `using`

```
class Square {  
    ...  
    using Shape::Shape;  
    ...  
}
```

# public, private, protected

- U deklaraciji razreda
  - `public`: dostupno iz svih dijelova programa
  - `protected`: dostupno razredu u kojem je definirano, razredu koji taj razred nasljeđuje i `friend` funkcijama i razredima
  - `private`: dostupno samo razredu u kojem je definirano i `friend` funkcijama i razredima



# public, private, protected

- Primjer za friend funkciju koja pristupa private članu

```
class Shape {  
    private:  
        int id;  
        friend void setId(Shape &s, int newId);  
        ...  
};
```

FriendFunction.cpp

```
void setId(Shape &s, int newId) { s.id = newId; }
```

```
int main (void) {  
    ...  
    setId(s, 10);  
    ...  
}
```

# public, private, protected

- Primjer za friend razred koji pristupa private članu

```
class Shape {  
    private:  
        int id;  
        friend class SomeClass;  
    ...  
};  
class SomeClass {  
    public:  
        void setId(Shape &s, int newId) { s.id = newId; }  
    ...  
}  
int main(void) {  
    Shape s(...);  
    SomeClass c(...);  
    c.setId(s, 10);  
    ...  
}
```

FriendClass.cpp

# public, private, protected

- Kod nasljeđivanja
  - `class Derived : public Base { ...`
    - `public` članovi iz `Base` postaju `public` članovi u `Derived`
    - `protected` članovi iz `Base` postaju `protected` članovi u `Derived`
    - `private` članovi iz `Base` nedostupni su u `Derived`
  - `class Derived : protected Base { ...`
    - `public` članovi iz `Base` postaju `protected` članovi u `Derived`
    - `protected` članovi iz `Base` postaju `protected` članovi u `Derived`
    - `private` članovi iz `Base` nedostupni su u `Derived`
  - `class Derived : private Base { ...`
    - `public` članovi iz `Base` postaju `private` članovi u `Derived`
    - `protected` članovi iz `Base` postaju `private` članovi u `Derived`
    - `private` članovi iz `Base` nedostupni su u `Derived`

# static članovi

- Svaki definirani objekt posjeduje vlastiti skup članskih varijabli i dijeli zajednički kôd članskih funkcija
- `static` članske varijable zajedničke su za sve instance razreda i postoje čak i ako nema instanciranih objekata
- automatski se inicijaliziraju na nulu, ili, ako su objekti, pretpostavljenim konstruktorom
- prije C++17 `static` članske varijable morale su se deklarirati u razredu ali i definirati izvan deklaracije razreda
- od C++17 `static` članske varijable mogu se definirati u deklaraciji razreda s `inline`

# static članovi

- `static` članske funkcije postoje čak i ako nema instanciranih objekata, nisu povezane niti s jednim objektom pa nemaju pristup pokazivaču `this` nego samo statičkim varijablama
- pozivaju se sintaksom *`razred::funkcija(argumenti);`*
- Primjer: skup likova s automatskim generiranjem id-a

**ShapeInheritanceWithStatic.cpp**

# Struktura (struct)

- Kao i u C-u, ali kod definicije nije potrebno struct

```
#include <iostream>
using namespace std;
struct Point {
    int x;
    int y;
};
int main(void) {
    Point p = {1, 2};
    Point r = {3};
    cout << p.x << " " << p.y << endl;
    cout << r.x << " " << r.y << endl;
    return 0;
}
```

Struct.cpp

1 2 ↵

3 0 ↵

# Struktura (`struct`)

- U C++ struktura može imati sve što i razred (članske funkcije, konstruktore, destruktore, nasljeđivanje...)
- Jedina razlika:
  - kod strukture su svi članovi, ako se drukčije ne navede, `public`
  - kod razreda su svi članovi, ako se drukčije ne navede, `private`
- Zbog semantike, preporuča se:
  - koristiti strukturu kad su u pitanju samo podaci
  - koristiti razred kad se definiraju i članske funkcije

# Višeznačnost operatora (*operator overloading*)

- izraz poput

*lijevi\_operand operator desni\_operand*

shvaća se kao članska funkcija razreda kojem pripada lijevi operand

*lijevi\_operand.funkcija\_tipa\_operator(desni\_operand)*

- Primjer:

`a = b + c;`     $\rightarrow$     `a = b.+(c);`

- Alternativno, isti izraz može se shvatiti kao ne-članska funkcija  
*funkcija\_tipa\_operator(lijevi\_operand, desni\_operand)*



# Višeznačnost operatora (*operator overloading*)

- Primjer: ostvariti razred `Fraction` (razlomak) i u njemu operatore zbrajanja i uspoređivanja

```
...  
class Fraction {  
    public:  
        int numerator;  
        int denominator;  
        Fraction(){};  
        Fraction(int numerator, int denominator)  
            : numerator(numerator), denominator(denominator) {}  
...  

```

**Fraction.cpp**

# Višeznačnost operatora (*operator overloading*)

...

```
Fraction operator+(const Fraction &other) {  
    Fraction r;  
    r.denominator = denominator * other.denominator;  
    r.numerator = denominator * other.numerator +  
                  other.denominator * numerator;  
    return r;  
}  
bool operator==(const Fraction &other) {  
    return denominator == other.denominator  
        && numerator == other.numerator;  
}
```

...

# Višeznačnost operatora (*operator overloading*)

- Alternativno

...

```
friend Fraction operator+(const Fraction &left,
                          const Fraction &right) {
    Fraction r;
    r.denominator = left.denominator * right.denominator;
    r.numerator = left.denominator * right.numerator +
                  right.denominator * left.numerator;
    return r;
}
friend bool operator==(const Fraction &left,
                       const Fraction &right) {
    return left.denominator == right.denominator &&
           left.numerator == right.numerator;
}
```

...

# Višeznačnost operatora (*operator overloading*)

- Za realizaciju

`cout << objekt_razreda_fraction`

ili

`cin >> objekt_razreda_fraction`

i oblik

*lijevi\_operand.funkcija\_tipa\_operator(desni\_operand)*

trebalo bi u razred ostream, odnosno istream dodati operatore kojima je *desni\_operand* tipa Fraction (i tako za svaki novi razred kojega se želi ispisati/učitati)

zato je jedino moguć oblik

*funkcija\_tipa\_operator(lijevi\_operand, desni\_operand)*

# Višeznačnost operatora (*operator overloading*)

```
...  
    friend ostream& operator<<(ostream &os, const Fraction &r) {  
        os << r.numerator << "/" << r.denominator;  
        return os;  
    };  
  
    friend istream &operator>>(istream &is, Fraction &r) {  
        is >> r.numerator >> r.denominator;  
        return is;  
    };  
...
```

# Višeznačnost operatora (*operator overloading*)

```
...
int main(void) {
    Fraction a(1, 2), b(2, 3), c(1, 2);
    cout << a << " " << b << " " << a + b << endl;
    cout << (a == b) << " " << (a == c) << endl;
    a = b;
    cout << a << " " << b << " "
         << (a == b) << " " << (a == c) << endl;
    return 0;
}
```

```
1/2 2/3 7/6 ↵
0 1 ↵
2/3 2/3 1 0 ↵
```

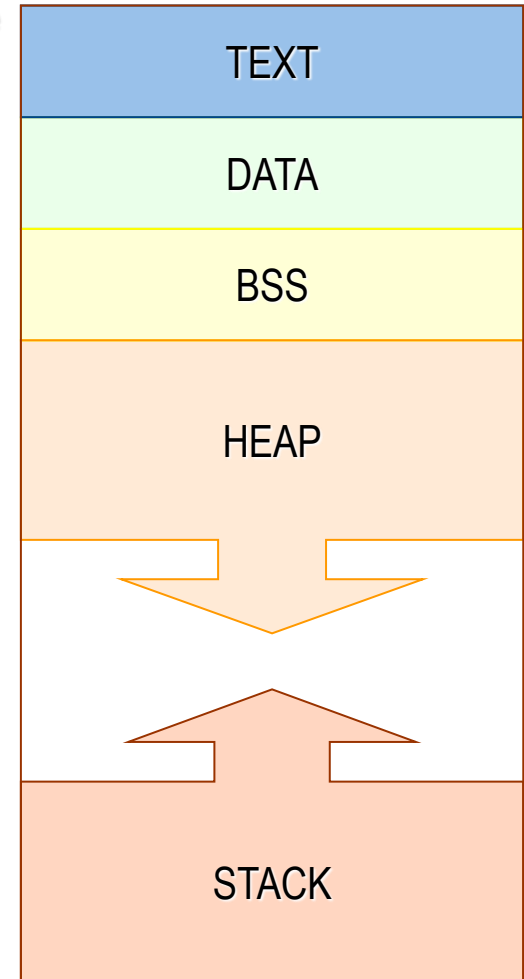
# Višeznačnost operatora (*operator overloading*)

- Operator pridruživanja nije potrebno definirati jer je ovdje prihvatljivo pretpostavljeno pridruživanje (memcpy)

# Dinamička rezervacija - segmenti memorije

- ovisi o operacijskom sustavu
  - TEXT
    - pohranjen program
  - DATA
    - inicijalizirane globalne i statičke lokalne varijable
  - BSS (Block Started by Symbol)
    - neinicijalizirane globalne i statičke lokalne varijable
  - gomila (heap)
    - dinamički rezervirana memorija
  - stog (stack)
    - lokalne varijable funkcija
    - nalazi se na dnu (najviše adrese)

niže adrese



više adrese



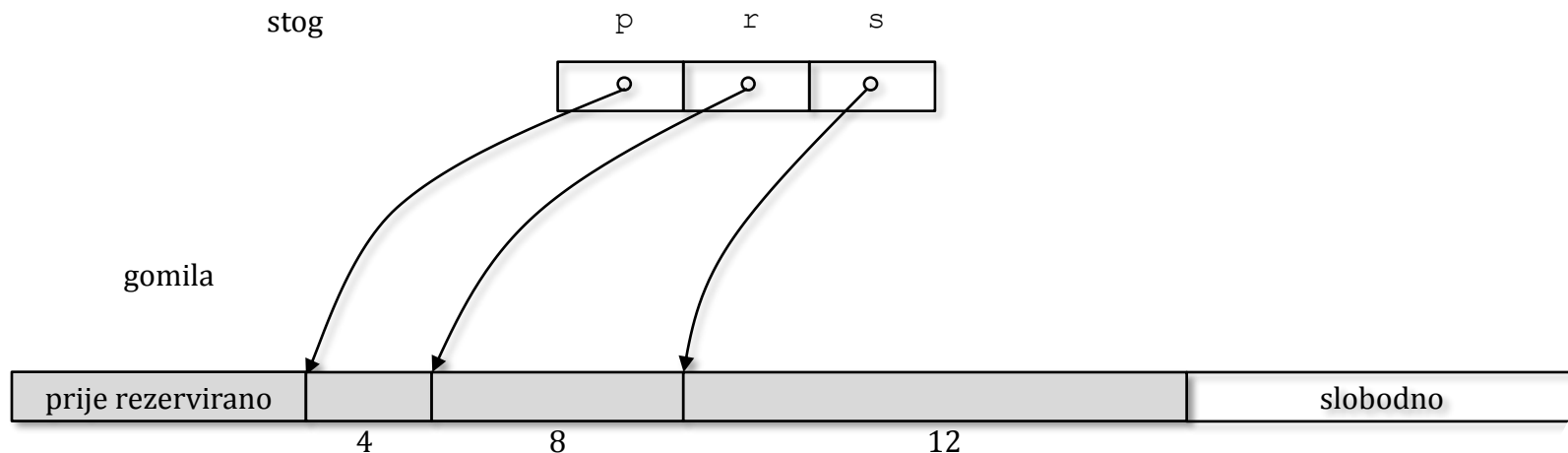
# Dinamička rezervacija memorije

- standardni C:
- `void *malloc(size_t size);`
  - pronalazi na gomili *size* slobodnih bajtova, rezervira ih i vraća vraća pokazivač na prvi od njih. Sadržaj tako rezervirane memorije ne inicijalizira se. Ako nema dovoljno memorije, vraća NULL.
- `void *calloc(size_t num, size_t size);`
  - pronalazi na gomili mjesto za *num* objekata pojedinačne veličine *size*, rezervira ih, inicijalizira sve bajtove unutar tako rezerviranog prostora na nulu, te vraća pokazivač na prvi od njih. Ako nema dovoljno memorije, vraća NULL.
- `void free(void *ptr);`
  - oslobađa memoriju na koju pokazuje *ptr* (koji mora biti neki od rezultata prethodnih poziva *malloc* ili *calloc*)

# Problem segmentacije memorije

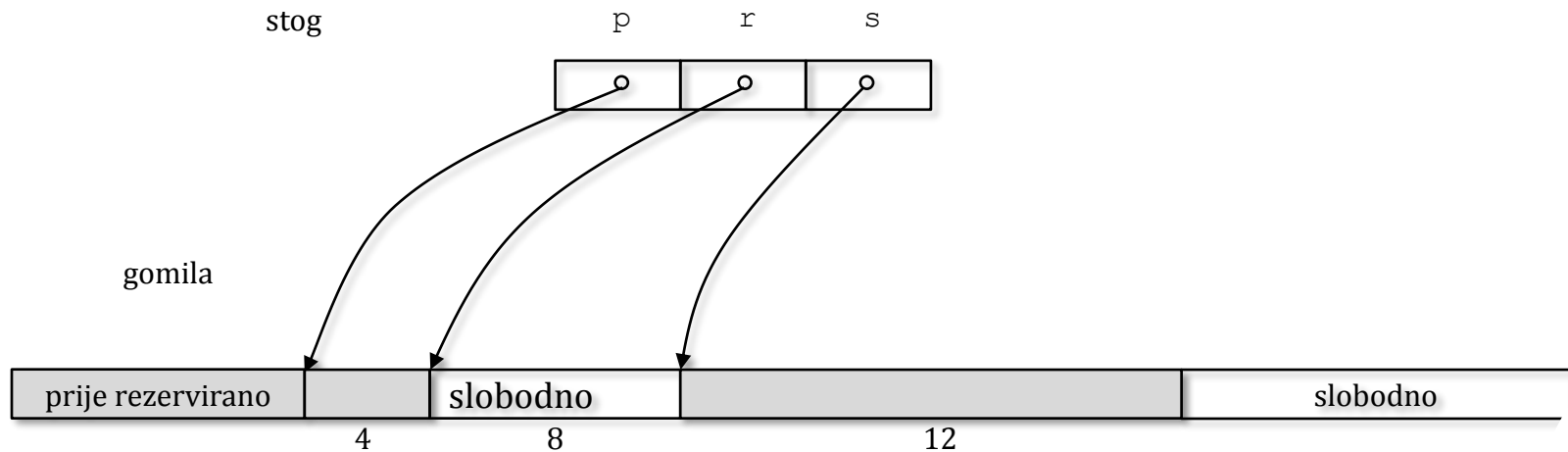
```
...  
char *p, *r, *s;  
p = (char *) malloc(4);  
r = (char *) malloc(8);  
s = (char *) malloc(12);
```

Shematska slika memorije ako na *heap-u* ima u kontinuitetu dovoljno mjesta



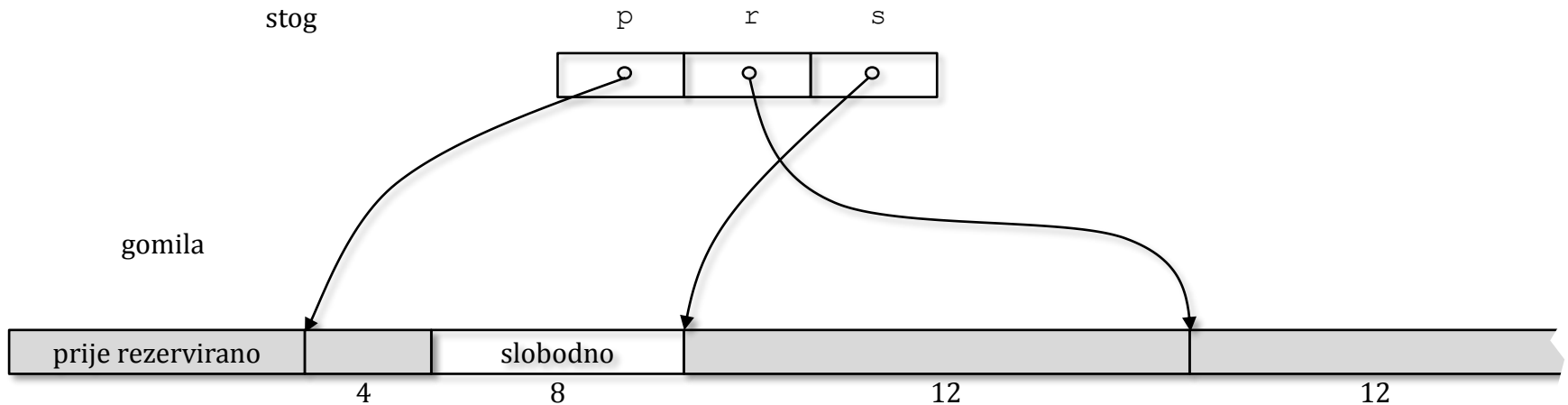
# Problem segmentacije memorije

```
...  
free(r);
```



# Problem segmentacije memorije

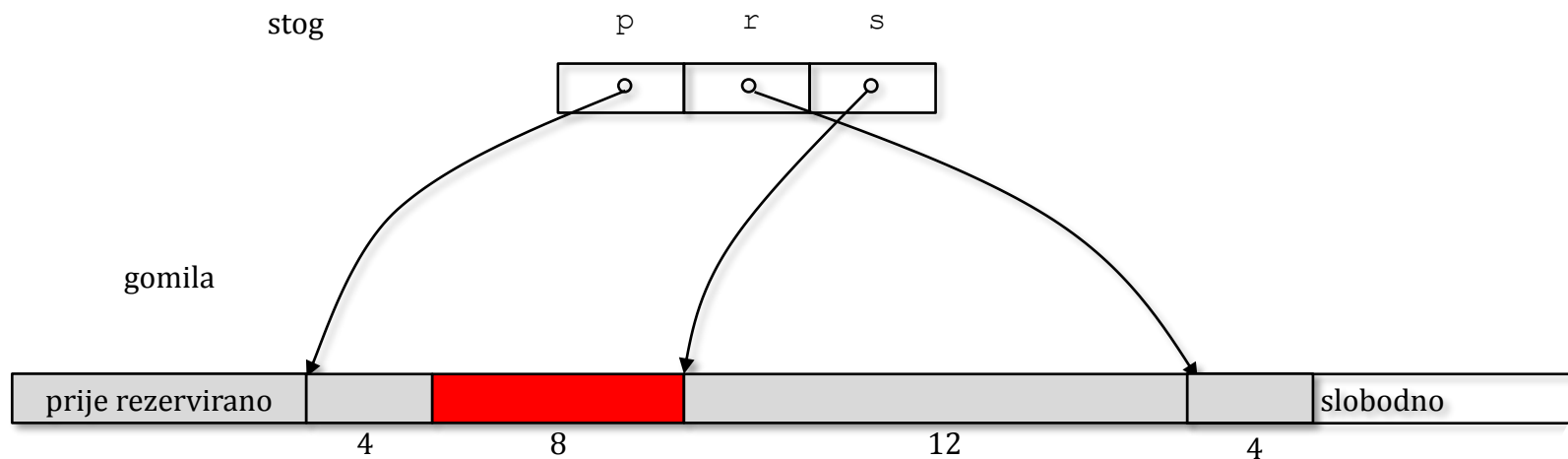
```
...  
r = (char *) malloc(12);
```



# "smeće" ili "curenje memorije" (*garbage, memory leak*)

...

```
char *p, *r, *s;  
p = (char *) malloc(4); r = (char *) malloc(8);  
s = (char *) malloc(12);  
r = malloc(4); // ili izlazak iz bloka
```



- ne postoji "*garbage collector*"!

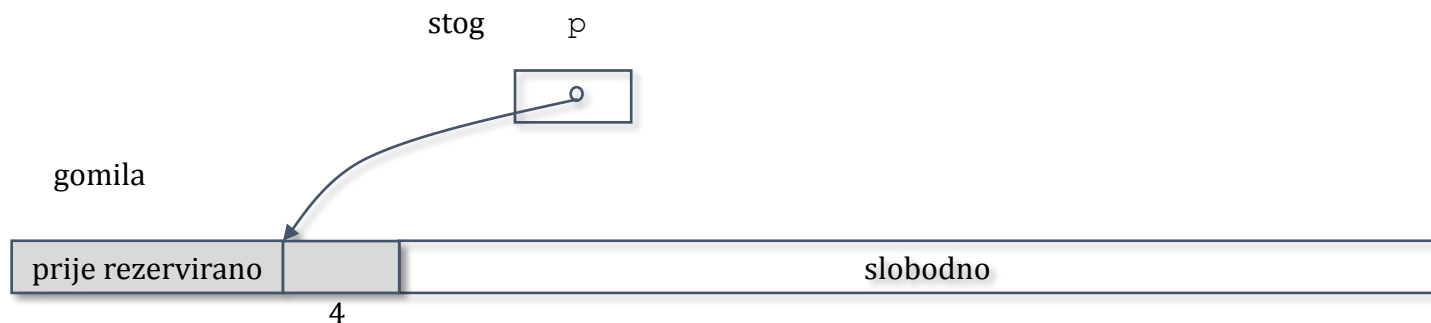
# Promjena rezervacije memorije

- `void *realloc(void *ptr, size_t newSize);`
- `ptr` može biti `NULL`, pri čemu `realloc` radi isto što i `malloc`
- inače, `ptr` mora biti rezultat prethodnog poziva `malloc` ili `realloc`

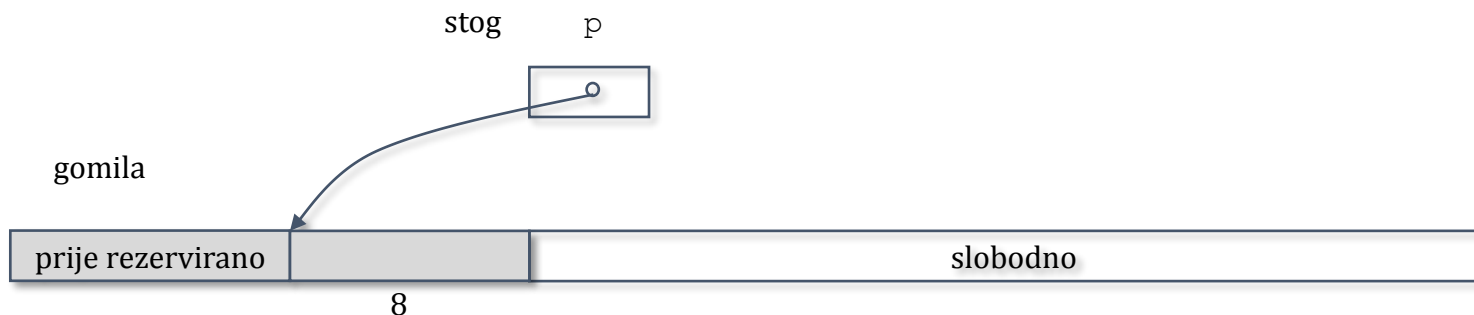
# Promjena rezervacije, mogući slučajevi

- Iza rezerviranog bloka ima dovoljno mjesta

```
p = malloc(4);
```



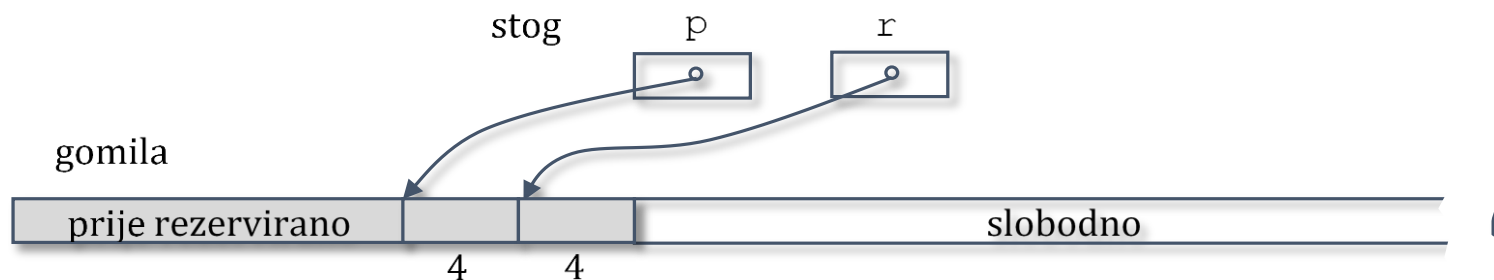
```
p = realloc(p, 8);
```



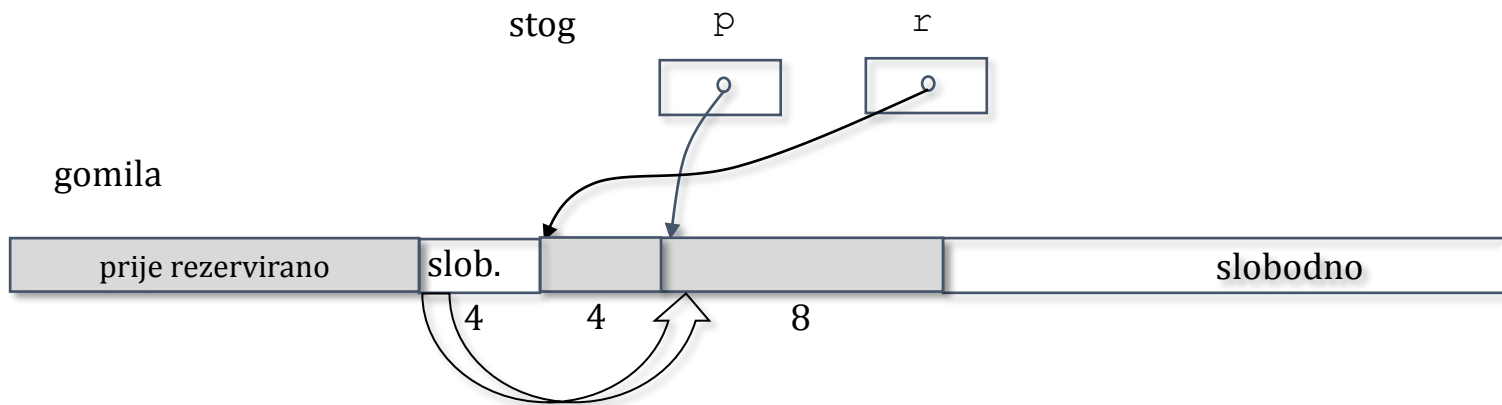
# Promjena rezervacije, mogući slučajevi

- Iza rezerviranog bloka nema dovoljno mjesta

```
p = malloc(4); r = malloc(4);
```



```
p = realloc(p, 8);
```





# nullptr

- u standardnom C-u NULL je integer:  
`#define NULL 0`
- C++ uvodi `nullptr` koji nije integer i ne može se pridružiti integeru
- korisno kod višeznačnosti funkcija

```
void f(int x) {  
    ...  
}  
void f(int *x) {  
    ...  
}
```
- `f(NULL)` pozvat će prvu (a po smislu bi se trebala pozvati druga)
- `f(nullptr)` pozvat će drugu

# Operatori new i delete

- `ptr = new tip;`
- ili
- `ptr = new tip(argumenti_konstruktor);`
  - pronalazi na gomili `sizeof(tip)` slobodnih bajtova, rezervira ih i vraća vraća pokazivač na prvi od njih. Ako rezervacija uspije, poziva se pretpostavljeni konstruktor ili konstruktor s odgovarajućim parametrima.
  - ako rezervacija ne uspije
    - aktivira se iznimka `std::bad_alloc`
    - ako se pozove s `new(nothrow)` vraća `nullptr`
- `delete ptr;`
  - oslobađa memoriju na koju pokazuje `ptr` i poziva destruktor, ako postoji

# new i delete - primjer

NewDelete.cpp

```
...
class C {
public:
    int x;
    C() : x(0) {}
    C(int x) : x(x) {}
    ~C() { cout << " ~"; }
};
...
```

# new i delete - primjer

NewDelete.cpp

```
...
int main(int argc, char *argv[]) {
    int *p;    C *r, *s;
    // s new je moguće rezervirati memoriju
    // i za osnovne tipove
    p = new int;
    // rezervacija uz poziv default konstruktora
    r = new C;
    // rezervacija uz poziv odgovarajućeg konstruktora
    s = new C(4);
    cout << *p << " " << r->x << " " << s->x;
    delete p;    delete r;    delete s;
    return 0;
}
```

-1163005939 0 4 ~ ~ ↵

# new i delete s poljem

- C++:
- `ptr = new tip[size];`
  - pronalazi na gomili `sizeof(tip) * size` slobodnih bajtova, rezervira ih i vraća vraća pokazivač na prvi od njih. Poziva pretpostavljeni konstruktor za svaki od elemenata polja. Nije moguće pozvati neki drugi konstruktor.
- `delete[] ptr;`
  - oslobađa memoriju na koju pokazuje `ptr` i poziva destruktor za svaki od elemenata polja
- ne postoji ekvivalent za `realloc`. `realloc` u slučaju potrebe seli sadržaj memorije na drugu lokaciju, a to predstavlja nerješiv problem kod složenih objekata koji u sebi referenciraju druge pokazivačima

# new i delete – primjer s poljem

NewDeleteArray.cpp

```
...
class C {
public:
    int x;
    C() : x(0) {}
    C(int x) : x(x) {}
    ~C() { cout << " ~"; }
};

int main(void) {
    C *r;
    r = new C[5];
    cout << r[0].x << " ";
    delete[] r;
    return 0;
}
```

0 ~ ~ ~ ~ ~ ↵

# Konstruktor kopiranjem (*copy constructor*)

- Što se dešava sljedećem primjeru?

...

```
void f(Fraction x) { ... }
```

...

```
Fraction a;
```

```
f(a);
```

...

- *Call by value* – stvaranje kopije objekta u funkciji
- U ovom primjeru sve je ok, jer će se napraviti memcopy

# Konstruktor kopiranjem (*copy constructor*)

- Ali što ako objekt, sadržava pokazivač na podatke koji se također žele kopirati, npr. razred String?

```
class String {  
    private:  
        char *p;  
        size_t size;  
    ...  
    String(String &s) { // konstruktor kopiranjem  
        p = (char *)malloc(s.size + 1);  
        size = s.size;  
        strcpy(p, s.p);  
    }  
    ...  
}
```



# Konstruktor kopiranjem i operatori kod složenijih razreda

- U C++ već postoji razred `string` koji ućahuruje sve radnje oko osnovnog podatka tipa `char *` (rezervacija memorije, pridruživanje, kopiranje, usporedba, ispis)
- Ali, instruktivna je vlastita realizacija razreda `String`  
`String.cpp`

# Rukovanje iznimkama (*exception handling*)

- try/catch blok

```
...  
try {  
    ...  
} catch (iznimka) {  
    ...  
}  
...
```

- ne postoji finally blok

# Rukovanje iznimkama

- Hijerarhija standardnih iznimaka
- <https://en.cppreference.com/w/cpp/error/exception>
- npr.
- `logic_error`
  - `invalid_argument`
  - `domain_error`
  - `length_error`
  - `out_of_range`
  - `future_error(C++11)`
- `throw length_error ("Tekst iznimke");`
- ali i
- `throw logic_error("Tekst iznimke");`

# Rukovanje iznimkama

- Primjer (samo za ilustraciju sintakse, *ne ovako programirati* 😊)

ExceptionHandling.cpp

```
#include <iostream>
using namespace std;
#define MAX 5
int main(void) {
    int p[5] = {1, 2, 3, 4, 5}, i;
    try {
        while (1) {
            cin >> i;
            if (i < 0 || i >= MAX)
                throw out_of_range("Indeks izvan granica");
            cout << p[i] << endl;
        }
    } catch (exception &e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

# Rukovanje iznimkama

- Vlastite iznimke

ExceptionHandlingOwn.cpp

```
...
int i;
while (1) {
    try {
        cin >> i;
        if (i == 0) throw i;
        else if (i == 1) throw "Text exception";
        else throw 2.2;
    } catch (int &ie) {
        cerr << "Integer exception " << ie << endl;
    } catch (const char *te) {
        cerr << te << endl;
    } catch (...) {
        cerr << "Neka druga iznimka" << endl;
    }
}
```

# Rukovanje iznimkama

## ■ Nasljeđivanje iznimaka

ExceptionHandlingInherit.cpp

```
...
class MyException : public runtime_error { // ima konstruktor s message
private:
    int number;
public:
    MyException(const char *message, int number)
        : runtime_error(message), number(number){};
    int code() { return number; }
};

int main(void) {
    try {
        throw MyException("Tekst", 3);
    } catch (MyException &ex) {
        cout << ex.code() << " " << ex.what() << endl;
    }
    return 0;
}
```

3 Tekst↵

# Primjer

- Dinamičko polje tipa `int` kojemu se pri stvaranju zadaje veličina, koja se kasnije može promijeniti

**DynamicArrayInt.cpp**

# Generičko programiranje

- C, C++: jezici s rigoroznom definicijom tipa (*strongly typed languages*)
- Varijable, parametri i rezultati funkcija moraju imati definiran tip
- Posljedica, standardni C, npr:

```
int abs(int arg);  
long labs(long arg);  
long long llabs(long long n);  
float fabsf(float arg);  
double fabs(double arg);  
long double fabsl(long double arg);
```

- `fabsl` bi mogla poslužiti umjesto svih ostalih, ali uz trošak konverzije pri pozivu i povratku



# Generičko programiranje

- Alternativa: makro

```
#define abs(arg) ((a) < 0 ? (-a) : (a))
```

- Ali, koliko je

```
int n = 3;  
n = abs(++n);
```

- I još:
  - Makro se uvijek razvija na mjestu poziva (kao i `inline` funkcija)
  - Neučinkovito i nepraktično za veće funkcije

# Generičko programiranje

- Predlošci (*templates*)
- Predložak funkcije

```
template <typename T>
T abs(T a) {
    return a < 0 ? -a : a;
}
```

- u konkretnom slučaju dobro je napraviti

```
template <typename T>
inline T abs(T a) {
    return a < 0 ? -a : a;
}
```

- razrješava se tijekom prevođenja

# Generičko programiranje

- Predložak razreda (*class template*)
- Najčešće za realizaciju zbirke ili kontejnera (u kasnijim poglavljima)
- Jednostavniji primjer

```
#include <iostream>
using namespace std;

template <typename T> class Point {
public:
    T x;
    T y;
    Point(T x, T y) : x(x), y(y) { }
};
...
```

TemplatePoint.cpp

# Generičko programiranje

```
int main(void) {  
    Point<int> pi(1, 2);  
    Point<double> pd(3.4, 4.5);  
    cout << pi.x << " " << pi.y << endl;  
    cout << pd.x << " " << pd.y << endl;  
    return 0;  
}
```

```
1 2  
3.4 4.5
```

# Generičko programiranje

- Složeniji primjer

```
#include <iostream>
using namespace std;
```

TemplatePairSimple.cpp

```
class Complex {
public:
    double real;
    double img;
    Complex() : real(0), img(0) {}
    Complex(double real, double img) : real(real), img(img) {}
};
...
```

# Generičko programiranje

```
template <class C> class Pair {
public:
    C a;
    C b;
    Pair(C a, C b) {
        this->a = a;
        this->b = b;
    }
};

int main(void) {
    Complex x(1, 2), y(3, 4);
    Pair<Complex> p(x, y);
    cout << p.a.real << " " << p.a.img << endl;
    cout << p.b.real << " " << p.b.img << endl;
    return 0;
}
```

1 2 ↩

3 4 ↩

# Generičko programiranje

- Još složeniji primjer: točka čije koordinate mogu biti proizvoljnog tipa

```
#include <iostream>
using namespace std;

template <typename T> class Point {
public:
    T x;
    T y;
    Point() : x(0), y(0) {}
    Point(T x, T y) : x(x), y(y) {}
};
...
```

TemplatePair.cpp

# Generičko programiranje

```
template <class C1, class C2> class Pair {
public:
    C1 a;
    C2 b;
    Pair(C1 a, C2 b) {
        this->a = a;
        this->b = b;
    }
};

int main(void) {
    Point<int> pi1(1, 2), pi2(3, 4);
    Point<double> pd1(1.1, 2.2), pd2(3.3, 4.4);
    Pair<Point<int>, Point<int>> ppi(pi1, pi2);
    Pair<Point<double>, Point<double>> ppd(pd1, pd2);
    cout << ppi.a.x << " " << ppi.a.y << endl;
    cout << ppd.a.x << " " << ppd.a.y << endl;
    return 0;
}
```



# Generičko programiranje

- Primjer za generički kontejner

**DynamicArray.cpp**

# Pametni pokazivači (*smart pointers*)

- ako se ne obavi *delete*, ostaje "smeće"

```
...
class SomeClass {
public:
    string x;
    SomeClass(string x) : x(x) {
        cout << "Object " << x << " created" << endl;
    }
    ~SomeClass() { cout << "Object " << x << " destroyed" << endl; }
};

void f() {
    SomeClass *p = new SomeClass("a");
    cout << p->x << endl;
}

int main(void) {
    f();
    return 0;
}
```

MemoryLeak.cpp

# Pametni pokazivači (*smart pointers*)

- pametni pokazivač uništava objekt na koji pokazuje, kada izađe iz dosega, npr. `unique_ptr`

```
...  
void f() {  
    unique_ptr<SomeClass> p(new SomeClass("a"));  
    cout << p->x << endl;  
}  
int main(void) {  
    f();    return 0;  
}
```

UniquePointers.cpp

# Pametni pokazivači (*smart pointers*)

- pametni pokazivač inicijalizira se
- u definiciji, npr.
  - `unique_ptr<SomeClass> p(new SomeClass("a"));`
  - potrebne su dvije sukcesivne rezervacije memorije, za objekt i potom za pokazivač
- funkcijom `make_unique`, npr.
  - `unique_ptr<SomeClass> p = make_unique<SomeClass> ("a");`
  - sav posao načini se jednom rezervacijom memorije
- objekt se eksplicitno uništava funkcijom `reset`
  - `p.reset();`
- funkcijom `reset` pametni se pokazivač može usmjeriti na drugi objekt
  - `p.reset (new SomeClass ("c"));`

# Pametni pokazivači (*smart pointers*)

- shared\_ptr omogućava da više pokazivača pokazuje na isti objekt, a da se objekt uništi tek kad nestane posljednja referenca

```
...
void f() {
    shared_ptr<SomeClass> p(new SomeClass("a"));
    cout << p->x << " " << p.use_count() << endl;
    p = make_shared<SomeClass>("b"); // uništava "a" i referencira "b"
    cout << p->x << " " << p.use_count() << endl;
    {
        shared_ptr<SomeClass> r = p;
        cout << p->x << " " << p.use_count() << endl;
        cout << r->x << " " << r.use_count() << endl;
    }
    cout << p->x << " " << p.use_count() << endl;
}
int main (void) { ...
```

SharedPointers.cpp

# C++ Standard Template Library (STL)

- Skup generičkih razreda koji implementiraju česte algoritme i strukture podataka (stogove, liste, polja itd.)
- Većina tih algoritama i struktura bit će detaljno obrađena na ovom predmetu
- 4 glavne komponente
  - Algoritmi (sort, search,...)
  - Kontejneri (razredi koji sadržavaju skup objekata npr. vector, list...)
  - Iteratori
  - Funkcijski objekti (*functors*)
    - objekti koji definiraju operator ()

# Iteratori

- Konzistentan način iteriranja kroz kontejner

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main(void) {
    vector<int> v = {1, 2, 3, 4, 5};
    vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;
    return 0;
}
```

Iterator.cpp

# Minimalna implementacija

- Mora se implementirati:
  - `begin()`
  - `end()`
- Mora još minimalno postojati (ili se implementirati)
  - `operator *()`
  - `operator ++()`
  - `operator !=()`
- Primjer:

**DynamicArrayGenericWithIterator.cpp**



# Funkcijski objekti

- Standardni C poznaje koncept pokazivača na funkciju
  - Ime funkcije predaje se kao argument drugoj funkciji
- Primjer: funkcija iz standardne biblioteke

```
void qsort(void *ptr, size_t count, size_t size,  
          int (*comp)(const void *, const void *));
```
- sortira bilo koje polje koristeći funkciju comp koja uspoređuje dva elementa
- Primjer:  
**PointerToFunction.cpp**
- Problem: takvoj funkciji ne može se predati argument tijekom poziva

# Funkcijski objekti

- Primjer: funkcija `for_each` iz `<algorithm>` primjenjuje zadanu funkciju nad svakim elementom kontejnera (koji mora imati implementiran iterator)
- Jednostavan primjer: treba pomnožiti s 2 svaki element

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
void print(int &i) { cout << i << " "; }
void multiply2(int &i) { i *= 2; }
int main(void) {
    vector<int> v = {1, 2, 3, 4, 5};
    for_each(v.begin(), v.end(), multiply2);
    for_each(v.begin(), v.end(), print);
    return 0;
}
```

FunctorMotivation.cpp

2 4 6 8 10

# Funkcijski objekti

- Ali kako pomnožiti s proizvoljnim faktorom ili ispisati po zadanom formatu?

```
#include <algorithm>
#include <stdio.h>
#include <string>
#include <vector>
using namespace std;
class Print {
public:
    // razred string se brine za odgovarajuću rezervaciju memorije
    string format;
    Print(string format) : format(format){};
    // ali printf treba char * koji vraća funkcija c_str()
    void operator()(int &i) { printf(format.c_str(), i); }
};
```

**Functor.cpp**

# Funkcijski objekti

```
class Multiply {  
    public:  
        int multiplier;  
        Multiply(int multiplier) : multiplier(multiplier){};  
        void operator()(int &i) { i *= multiplier; }  
};  
  
int main(void) {  
    vector<int> v = {1, 2, 3, 4, 5};  
    for_each(v.begin(), v.end(), Multiply(3));  
    for_each(v.begin(), v.end(), Print("%03d "));  
    return 0;  
}
```

003 006 009 012 015 ↵