

# Informatika 3

## 5

### Objektovo-orientované prvky C++

# Trieda

- môžeme si predstaviť ako štruktúru, ktorá v sebe zahŕňa:
  - atribúty (dátové členy)
  - metódy s nimi narábajúce
- definuje sa **struct** alebo **class** (aj union)

# Prístupové kvalifikátory

- definuje kto môže použiť členy triedy
- platí pre atribúty aj metódy
- Používajú sa ako návestia – v rámci triedy kvalifikátor platí pokiaľ neuvedieme iný
- private - prístupné len pre metódy a nie pre zvyšok programu, public - prístupné pre všetkých
- rozdiel medzi class a struct:
  - struct má implicitne všetko public
  - class má implicitne všetko private

```

class C
{
    int data;
    char meno[11];
    int GetData() { return data;}
    char* GetMeno() { return meno;}
};

struct S
{
    int data;
    char meno[11];
    int GetData() { return data;}
    char* GetMeno() { return meno;}
};

union U
{
    int data;
    char meno[11];
    int GetData() { return data;}
    char* GetMeno() { return meno;}
};

```

```

int main()
{
    C oc;
    oc.GetData(); // Nepripustne
    oc.GetMeno(); // Nepripustne

    S os;
    os.GetData(); // OK
    os.GetMeno(); // OK

    U ou;
    ou.GetData(); // OK
    ou.GetMeno(); // OK
}

```


# Dátové členy - atribúty

## Pre atribúty platí :

- môžu to byť rôznych typov, základných resp. tried (predtým definované)
- atribút nemôže byť objekt definovanej triedy, ale môže byť smerník (referencia)

```
/* NIE */  
class Trieda {  
    Trieda atribut;  
};
```

```
/* ANO */  
class Trieda {  
    Trieda* atribut;  
};
```



# Metódy

- právoplatné činnosti s atribútmi triedy
- pre metódy platí to isté čo pre nečlenské (globálne) funkcie
- metódy môžu byť definované buď v definícii triedy, alebo mimo - vtedy majú inú hlavičku  
**navratova\_hodnota trieda::funkcia(parametre);**
- metódy definované priamo v triede sú **inline**
  - pre jednoduché funkcie
  - funkcia nie je volaná ako štandardná funkcia ale je kompilovaná priamo do volania

```
struct Auto {  
    int aPocKolies;  
    int GetPocKolies() { return aPocKolies; }  
    inline  
    void SetPocKolies( int x );  
};
```

auto.h

```
void Auto::SetPocKolies( int x )  
metóda  
{  
    aPocKolies = x;  
}
```

auto.cpp

# Objekt - Inštancia triedy

- konkrétna inštancia triedy - fyzicky existuje v pamäti
- prístup k položkám objektu (atribútom aj metódam) rovnaká ako v C pre prístup k položkám štruktúry struct – operátor ‘.’ resp. ‘->’
- môžeme si predstaviť ako premennú typu trieda

```
Auto skoda; // Auto - trieda, skoda - objekt
cout << obj.GetPocKolies(); // volanie metódy
Auto::GetPocKolies
obj.SetPocKolies(10); // volanie metódy
Auto::SetPocKolies
```

# Príklad

cAuto.h

```
enum Farba {Biela, Cierna, Cervena, Zlta};

class Auto {
private: // atribúty
    int    aNumCes = 0; /* pocet cestujucich */
    int    aMaxCes = 0; /* pocet miest */
    eFarba aFarba;
public: // metódy
    void    PridajCestujuci(int a);
    void    ZrusCestujuci(int a);
    void    NastavFarbu(eFarba f) {aFarba = f;}
};
```

Auto.cpp

```
void Auto::PridajCestujuci(int x)
{
    if(aNumCes+x <= aMaxCes) aNumCes +=
x;
}
void Auto::ZrusCestujuci(int x)
{
    if(aNumCes-x >= 0) aNumCes -= x;
}
```

hlavny.cpp

```
void main()
{
    cAuto skoda;

    skoda.NastavFarbu(Cervena)
;
    skoda.PridajCestujuci(3);
    skoda.ZrusCestujuci(2); 7
}
```



# Ako funguje volanie metódy

- pri volaní metódy má každá presne definované pre ktorý objekt bola volaná tzv. implicitný objekt, preto netreba zložitým spôsobom odkazovať na atribúty a metódy objektu
- v praxi je to realizované tak, že sa funkcií predáva ešte jeden (nultý) parameter, ktorý je smerníkom na daný objekt
- tento parameter môžeme použiť aj v metóde (kľúčové slovo **this**)
- metódy sú rovnocenné:

```
void Auto::PridajCestujuci(int x)
{
    if(aNumCes+x <= aMaxCes)
        aNumCes += x;
}
```

```
void Auto::PridajCestujuci(Auto* this, int x)
{
    if(this->aNumCes+x <= this->aMaxCes)
        this->aNumCes += x;
}
```

→ skrytý parameter

# Konštruktor

- špeciálna metóda, ktorá sa volá pri vzniku objektu - využíva sa a na inicializáciu objektu
- meno konštruktora je zhodné s menom triedy do ktorej patrí
- nesmie mať návratovú hodnotu
- objekt môže mať niekoľko konštruktorov, ktoré sa rozlišujú argumentmi (ako u preťažených funkcií)
- ak pre triedu nenadefinujeme konštruktor, kompilátor vygeneruje tzv. štandardný (implicitný) konštruktor - nemá žiadny argument a nevykonáva žiadnu činnosť
- ak nadefinujeme nejaký konštruktor, implicitný sa nevygeneruje a my ho musíme definovať explicitne
- argumenty konštruktora môžu byť aj implicitné

```
complex(double r, double i)  
{ real = r; imag=i; }
```

# Implicitné hodnoty argumentov

- v C++ môžeme volať funkciu s menším počtom argumentov, ak sú tieto implicitne definované v definícii funkcie
- implicitná hodnota môže byť:
  - globálna konštanta
  - globálna premenná
  - volanie funkcie
- môžu byť iba v prototypoch, nie v definíciách funkcií
- implicitné argumenty musia nasledovať za sebou, musia byť zoskupené dohromady a musia byť ako posledné argumenty

**void ProcNespravna(int a=1, int b, int c=3, int d=4);** // toto je chybný prototyp

**void ProcSpravna(int a, int b=2, int c=3, int d=4);** // a tento je správny

**ProcSpravna(10,15,20,25);** // OK: argumenty pre všetky parametre  
**ProcSpravna();** // CHYBA: parameter 'a' nemá implicitný argument

**ProcSpravna(12,15);** // OK: param. 'c' a 'd' sa priradí implicitná hodnota

**ProcSpravna(3,10,,12);** // CHYBA: vynechané parametre musia nasledovať

// tesne po sebe

# Implicitné hodnoty argumentov

## Príklad

```
class complex
{
private:
    double real;
    double imag;
public:
    complex(double r = 0, double i = 0)
    {
        real = r;
        imag = i;
    }
    void set(double r, double i = 0)
    {
        real = r;
        imag = i;
    }
};
```

# Konverzný konštruktor

- konštruktor aspoň s jedným parametrom
- jednosmerná konverzia z iného typu na objekt danej triedy
- volá sa automaticky

```
complex::complex(double i)
{
    real = i;
    imag = 0.0;
}
```

```
complex a;
a = 3;
```

```
explicit complex(double r=0, double i=0);
```

# Kopírovací konštruktor

- pri vytváraní jedného objektu z druhého sa používa tzv. copy-konštruktor (napr. pri výraze  $c = a + b$  kde  $a$ ,  $b$ ,  $c$  sú objekty, alebo pri predávaní objektu funkcii hodnotou)
- má jeden parameter - odkaz na objekt rovnakej triedy
- ak ho nenadefinujeme, prekladač ho vygeneruje sám (kopíruje bit po bite) - pozor, ak sa používajú v objekte ako pamäťové prvky smerníky

```
complex(const complex &x)  
{  
    real = x.real;  
    imag = x.imag;  
}
```

# Kopírovací konštruktor

```
#pragma once
#include <iostream>
#include <vector>
using namespace std;

class Copy
{
private:
    // Smernik na int
    int* data = nullptr;
public:

    Copy(int d) // Konštruktor
    {
        data = new int; // Alokuj objekt v dynamickej pamati
        *data = d;
        cout << "Konštruktor:\t\t\t\t" << *data << "\t[" << data << "]" << endl;
    };

    Copy(const Copy& zdroj) // Kopirovaci konštruktor
    {
        data = new int; // Kopirovanie dat - hlboka kopia
        *data = *zroj.data;
        cout << "Kopirovaci konštruktor - hlboka kopia:\t" << *data << "\t[" << data << "]" << endl;
    }

    ~Copy() // Destruktor
    {
        if (data != nullptr) // Ak smernik neukazuje na nullptr
            cout << "Destruktor:\t\t\t\t" << *data << "\t[" << data << "]" << endl;
        else // Ak je smernik nullptr
            cout << "Destruktor pre nullptr" << endl;
        delete data; // Dealokuj pamat
    }
};
```

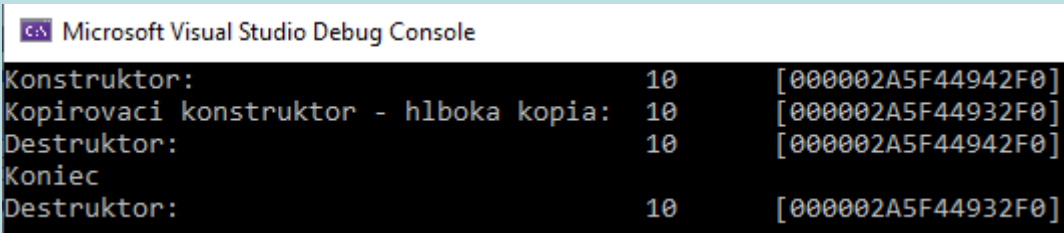
# Kopírovací konstruktor

```
// Kopirovací konstruktor
Copy(const Copy& zdroj)
: Copy(*zdroj.data) // Copy {*zdroj.data}
{
    // Kopírování dat - hluboká kopie
    cout << ++Citac << ". " << "Kopirovací konstruktor - hluboká kopie: " << *zdroj.data <<
endl;
}

#include "Copy.h"

int main()
{
    vector<Copy> vec;

    vec.push_back(Copy(10));
    cout << "Koniec" << endl;
    return 0;
}
```



Microsoft Visual Studio Debug Console

Konstruktor:	10	[000002A5F44942F0]
Kopirovací konstruktor - hluboká kopie:	10	[000002A5F44932F0]
Destruktor:	10	[000002A5F44942F0]
Koniec		
Destruktor:	10	[000002A5F44932F0]



# Presúvací konštruktor

- Presúva dynamicky alokované zdroje objektu do iného objektu
  - Trieda nemá používateľom definovaný kopírovací konštruktor
  - Trieda nemá používateľom definovaný priradovací operátor
  - Trieda nemá používateľom definovaný presúvací operátor
  - Trieda nemá používateľom definovaný deštruktor
  - Presúvací konštruktor nie je implicitne definovaný ako deleted

# Presúvací konštruktor

```
class Move {
private:
    int* data; // Smernik na int
public:
    Move(int d) // Konštruktor
    {
        data = new int; // Alokuj objekt v dynamickej pamati
        *data = d;
        cout << "Konštruktor:\t\t\t" << *data << "\t[" << data << "]" << endl;
    };

    Move(const Move& zdroj) // Kopirovací konštruktor
    {
        data = new int; // Kopirovanie dat - hlboka kopia
        *data = *zdroj.data;
        cout << "Kopirovací konštruktor - hlboka kopia:\t" << *data << "\t[" << data << "]" << endl;
    }

    Move(Move&& zdroj) noexcept // Presuvací konštruktor
        : data(zdroj.data) //: data{ zdroj.data }
    {
        cout << "Presuvací konštruktor:\t\t\t" << *data << "\t[" << data << "]" << endl;
        zdroj.data = nullptr;
    }

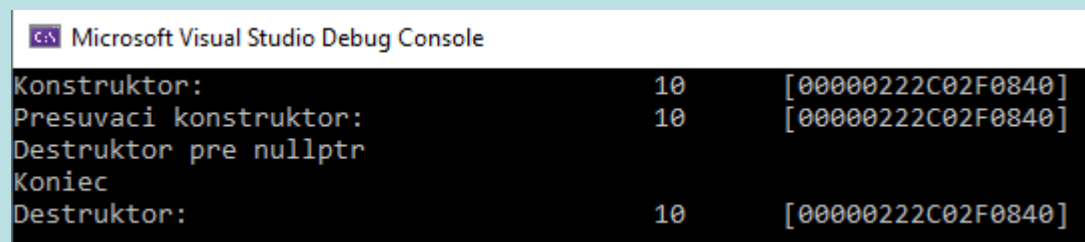
    ~Move() // Destruktor
    {
        if (data != nullptr) // Ak smernik neukazuje na nullptr
            cout << "Destruktor:\t\t\t\t" << *data << "\t[" << data << "]" << endl;
        else // Ak je smernik nullptr
            cout << "Destruktor pre nullptr" << endl;
        delete data; // Dealokuj pamat
    }
};
```

# Presúvací konštruktor

```
// Kopirovací konstruktor
Move(const Move& zdroj)
: Move{ *zdroj.data }
{
// Kopirovanie dat - hlboka kopia
cout << "Kopirovací konstruktor - Hlboka kopia: " << *data << "[" << data << "]" <<
endl;
}
```

```
#include "Move.h"
```

```
int main()
{
    vector<Move> vec;
    vec.push_back(Move(10));
    cout << "Koniec" << endl;
    return 0;
}
```



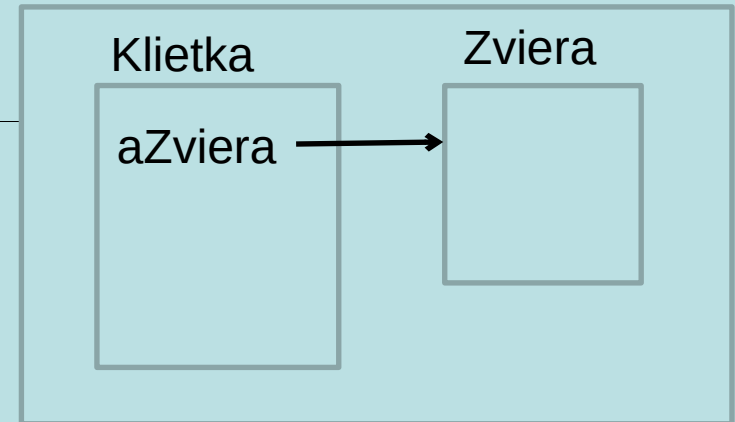
Microsoft Visual Studio Debug Console

Konstruktor:	10	[00000222C02F0840]
Presuvaci konstruktor:	10	[00000222C02F0840]
Destruktor pre nullptr		
Koniec		
Destruktor:	10	[00000222C02F0840]

# Deštruktor

- je špeciálna metóda, ktorá sa volá tesne pred uvoľnením objektu (opak ku konštruktoru)
- meno deštruktora je rovnaké ako meno triedy iba pred ním stojí tilda (~)
- nemá argumenty ani návratovú hodnotu
- trieda môže mať iba jeden deštruktor

```
class Zviera{  
};  
  
class Klietka{  
    Zviera* aZviera;  
public:  
    Klietka() { aZviera=new Zviera(); } //vytvorenie objektu  
Zviera  
    ~Klietka() { delete aZviera; } //uvolnenie objektu zviera  
};  
  
void Procedura()  
{  
    Klietka k;//vytvorenie objektu klietka - konštruktor  
    //...  
} //pri ukončení procedúry uvoľnenie lokálnych objektov -  
deštruktor
```



# Pret'azovanie (overloading)

- C++ dovoľuje aby existovalo viac funkcií a operátorov rovnakého mena, ktoré sa odlišujú argumentmi
- v C musíme mať pre funkciu abs inak pomenovanú funkciu pre každý typ

```
// v C
int    abs(int i);
long   labs(long l);
double fabs(double d);

// v C++
int    abs(int i);
long   abs(long l);
double abs(double d);

// volanie
abs(-10);      // volá int abs(int)
abs(-100000); // volá long abs(long)
abs(-10.34);   // volá double abs(double)
```

# Pret'azovanie (overloading)

- ak existuje pret'azená funkcia s identickými typy parametrov, zavolá sa táto implementácia
- ináč C++ kompilátor zavolá tú pret'azenú funkciu, ktorá zabezpečuje najľahšiu sériu konverzií

```
int    abs(int i);  
long   abs(long l);  
double abs(double d);  
  
abs('a');      // volá int abs(int i)  
abs(3.1415F);  // volá double abs(double d)
```

# Obmedzenia preťažovaných funkcií

- Preťažené funkcie sa musia odlišovať v type alebo počte parametrov. Nie iba návratovou hodnotou
- Typy parametrov musia byť naozaj rozdielne (typedef je iba nové pomenovanie typu)  

```
typedef INT int;  
// CHYBA obidva prototypy majú identické použitie  
void Nejednoznacna(int x);  
void Nejednoznacna (INT x);
```
- Na rozlíšenie typov parametrov funkcie môžeme použiť i kvalifikátor *const*  

```
void Proc(int ch);  
void Proc(const int ch);  
int main()  
{  
    const int c1 = 10;  
    int c2 = 'b';  
    Proc(c1); // volá sa void Proc(const int ch);  
    Proc(c2); // volá sa void Proc(int ch);  
}
```
- Preťažené funkcie by mali vykonávať filozoficky príbuzné činnosti. Vytvoriť implementáciu funkcie *abs*, ktorá by vracala druhú odmocninu čísla by bolo asi hlúpe a mätúce.<sup>9</sup>

# Operátorové metódy

- preťažovanie existujúcich operátorov
- ľavý operand je objekt

```
class complex {  
    double real, imag;  
public:  
    complex(double r=0,double i=0)  
        { real=r; imag=i; }  
    complex operator+(complex a)  
        { return complex(real+a.real, imag+a.imag); }  
    double operator++()// prefixovy  
        { real+=1; return real; }  
    double operator++(int)// postfixovy  
        { double old=real; real+=1; return old; }  
};  
  
complex x(1,2), y(3,4), z(5,6);  
z=x+y;//volanie operátora klasicky  
x=x.operator+(y);//volanie operátora metódou
```



# Prirad'ovací kopírovací operátor

- ak tento operátor nedefinujeme a je potrebný, prekladač si vygeneruje sám implicitný (kopíruje bit po bite)
- definuje sa podobne ako copy-konštruktor. Líši sa len v tom, že nevytvára nový objekt, ale modifikuje už existujúci

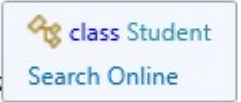
```
complex complex::operator = (const complex &zdroj)
{
    if(this == &zdroj) return *this;
    real = zdroj.real;
    imag = zdroj.imag;
    return *this;
}
```

- dôležité testovať samopriradenie - pri zložitých objektoch môže samopriradenie viesť ku katastrofe
- Každý operátor \*=, +=, >>=, <<=, ... je nutné definovať zvlášť

# Kopírovací konštruktor a prirad'ovací kopírovací operátor

```
#pragma once

class Student
{
private:
    char* meno = nullptr;
    char* priezvisko = nullptr;
    unsigned int priemer = 0;
public:
    Student(const char* pmeno, const char* ppriezvisko, unsigned int ppriemer = 0);
    Student(const Student& zdroj);
    ~Student()
    {
        delete[] priezvisko;
        delete[] meno;
        priemer = 0;
    }
    Student& operator =(const Student& zdroj);
private:
    char* KopirujRetazec(const char* pZdrojRetazec);
};
```

A tooltip box is visible over the code, containing the text "class Student" and a link "Search Online".

# Kopírovací konštruktor a prirad'ovací kopírovací operátor

Správne

```
Student::Student(const char* pmeno, const char* ppriezvisko, unsigned int ppriemer)
: priemer(ppriemer)
{
    if (pmeno && *pmeno)
    {
        int pocetZnakov = strlen(pmeno);
        meno = new char[pocetZnakov + 1];
        strcpy(meno, pmeno);
    }
    if (ppriezvisko && *ppriezvisko)
    {
        int pocetZnakov = strlen(ppriezvisko);
        priezvisko = new char[pocetZnakov + 1];
        strcpy(priezvisko, ppriezvisko);
    }
}
```

```
char* Student::KopirujRetazec(const char* pZdrojRetazec)
{
    char* kopiaRetazec = nullptr;
    if (pZdrojRetazec && *pZdrojRetazec)
    {
        int pocetZnakov = strlen(pZdrojRetazec);
        kopiaRetazec = new char[pocetZnakov + 1];
        strcpy(kopiaRetazec, pZdrojRetazec);
    }
    return kopiaRetazec;
}
```

Správne - lepšie

```
Student::Student(const char* pmeno, const char* ppriezvisko, unsigned int ppriemer)
: priemer(ppriemer)
{
    meno = KopirujRetazec(pmeno);
    priezvisko = KopirujRetazec(ppriezvisko);
}
```

# Kopírovací konštruktor a prirad'ovací kopírovací operátor

## Nesprávne

```
Student::Student(const Student& zdroj)
    : meno(zdroj.meno), priezvisko(zdroj.priezvisko), priemer(zdroj.priemer)
{
}

Student::Student(const Student& zdroj)
{
    meno = zdroj.meno;
    priezvisko = zdroj.priezvisko;
    priemer = zdroj.priemer;
}
```

## Správne

```
Student::Student(const Student& zdroj)
    : priemer(zdroj.priemer)
{
    meno = KopirujRetazec(zdroj.meno);
    priezvisko = KopirujRetazec(zdroj.priezvisko);
}

Student::Student(const Student& zdroj)
    : Student{ zdroj.meno, zdroj.priezvisko, zdroj.priemer }
{
}
```



# Kopírovací konštruktor a prirad'ovací kopírovací operátor

## Nesprávne

```
Student& Student::operator =(const Student& zdroj)
{
    priemer = zdroj.priemer;
    meno = zdroj.meno;
    priezvisko = zdroj.priezvisko;
    return *this;
}
```

## Správne

```
Student& Student::operator =(const Student& zdroj)
{
    if (&zdroj != this)
    {
        Student::~~Student();
        priemer = zdroj.priemer;
        meno = KopirujRetazec(zdroj.meno);
        priezvisko = KopirujRetazec(zdroj.priezvisko);
    }
    return *this;
}
```



# Prirad'ovací presúvací operator

Generuje sa ak nie je používateľom definovaný presúvací operátor priradenia a platí:

- Trieda nemá používateľom definovaný kopírovací konštruktor
- Trieda nemá používateľom definovaný presúvací konštruktor;
- Trieda nemá používateľom definovaný kopírovací prirad'ovací operátor
- Trieda nemá používateľom definovaný deštruktor

```
complex complex::operator =(const complex&& zdroj)
```

# Operátorové funkcie

- C++ dovoľuje definovať aj operátorové funkcie

```
struct compl { double real; double imag; };
```

```
compl operator+(compl a, compl b)
{
    compl ret;
    ret.real = a.real + b.real;
    ret.imag = a.imag + b.imag;
    return ret;
}
```

```
compl x, y, z;
x.real=3; x.imag=4;
y.real=30; y.imag=40;
```

```
z = x + y;           // volá sa compl operator+(compl a, compl b)
// alebo
z = operator+( x, y ) // funkčný zápis operátora
```

- Operátor je možné volať operátorovým zápisom alebo funkčným zápisom – rovnocenné
- Funkčný zápis sa využíva pri nejednoznačnosti
- operátor = môže byť iba operátorová metóda ale nie operátorová funkcia

# Kľúčové slovo friend

- umožňuje nečlenskej funkcii alebo objektu mať prístup k privátnym členom triedy

```
class Vec {  
    private:  
        int data;  
    public:  
        friend void nacistaj(Vec t, int x); // nie je metóda !!!  
};  
  
void nacistaj(Vec t, int x)  
{  
    t.data = x;  
}
```

---

```
class kohut;  
class kurca {  
    public:  
        friend class kohut;  
};
```

```
class kohut {  
    // čokoľvek  
};
```



# Friend funkcie a operátory

- často sa binárne komutatívne operátory nedefinujú ako metódy ale ako globálne operátory
- aby mali prístup k privátnym členom, musíme ich definovať ako friend

```
class complex {  
    double real, imag;  
public:  
    complex(double r, double i) { real=r; imag=i; }  
    friend complex operator+ (const complex&, const complex&);  
};
```

```
complex operator+ (const complex& a, const complex& b)  
{  
    complex c;  
    c.real  = a.real  + b.real;  
    c.imag = a.imag + b.imag;  
    return c;  
}
```

```
complex z(10,20), c;  
c = 10 + z;  // bude to fungovať ???
```

# Konverzie

- pre triedu si môžeme nadefinovať konverzie:
  - z iného typu – pomocou konverzných konštruktorov
  - na iný typ – pomocou konverzných operátorov
- konverzný konštruktor - konvertuje hodnotu iného typu na objekt danej triedy

```
complex::complex(int i)
{
    real = (double)i;
    imag = 0.0;
}
```

- konverzný operátor - konvertuje objekt danej triedy na nejaký typ

```
operator typ( ) { ... } // formálne nemá uvedený návratový typ!!!

operator double ( )
{
    return real;
}
```

# Príklad konverzií

```
class integer {  
private:  
    int value;  
public:  
    // kombinovaný normálny&konverzný konštruktor  
    integer(int i=0) {value = i; };  
    // konverzný operátor  
    operator int( ) {return value; };  
    // priradovací operátor  
    void operator = (const integer &zdroj) {value = zdroj.value;}  
    friend integer operator+(integer p, integer q);  
};  
integer operator+(integer p, integer q) { return integer(p.value + q.value);}
```

integer z;	// integer(0)
integer x(10);	// integer(10)
integer y = 25;	// integer(25) !!!
int a = int(x);	// operator int()
z = a;	// temp = integer(a); operator= (temp)
x = z;	// operator = (z);
a = z;	// a = z.operator int( ); !!!
int b = a + x;	// ???

# Obmedzenia preťažovania operátorov

- Je možné preťažiť všetky operátory okrem operátorov:

`:: . .* ?:`

- Priorita operátorov ostáva

```
class compl { double real, imag;
public:
    compl(double r, double i) { this->real=r; this->imag=i; }
    compl operator++()
    {
        this->real+=1;
        this->imag+=1;
        return x;
    }
    compl operator++(int) // postfixovy (použitý odkaz &)
    {
        compl old=x;
        this->real+=1;
        this->imag+=1;
        return old;
    }
};
compl c; c.real=2; c.imag=3;
// c=(4,5) d=(3,4) e=(3,4)
```

- Operátory by mali vykonávať filozoficky príbuzné činnosti k pôvodným operátorom

# Typovo bezpečné linkovanie

- C++ dotvára mená funkcií tak, že im pridáva za názov znaky reprezentujúce typ parametrov
- ak chceme použiť funkcie vytvorené v C musíme prototypy deklarovať takto:

```
extern "C" {  
    double sin(double x);  
    double cos(double x);  
    double tan(double x);  
}
```

# Entropia softvéru



- Nenechať zlý návrh, nesprávne rozhodnutia, chybný alebo nekvalitný kód neopravený
- Zadebniť, zabrániť škodám (výnimka, neimplementované, zakomentovať, ...)
- Nedovoliť entropii zvíťaziť

*Nenechávať  
„rozbité okná“*

