Objekty a třídy

Klasický program: samostatně definovány proměnné + samostatně definovány funkce.

Objektově orientovaný program: proměnné a funkce jsou definovány spolu v objektech.

Třída – definuje strukturu objektů.

Objekt – je instance třídy.

Třída je strukturovaný datový typ obsahující nejen proměnné, ale i funkce.

```
float a1=3,a2=5;

float obsahCtverce(float a) { return a*a; }

obsahCtverce(a1)     // obsah prvního čtverce

obsahCtverce(a2)     // obsah druhého čtverce
```

```
struct Ctverec { float a;
                 float obsah() { return a*a; }
};
Ctverec c1,c2;

c1.a=3; c2.a=5;

c1.obsah()     // obsah prvního čtverce

c2.obsah()     // obsah druhého čtverce
```

Specifické funkce třídy - konstruktor a destruktor

Vlastnosti funkce konstruktoru:

- Má stejné jméno jako třída.
- Nevrací hodnotu a ani se před jeho jménem neuvádí klíčové slovo *void* .
- Je polymorfní funkce – můžeme mít více konstruktorů lišící se typem nebo počtem parametrů.
- Uvedení konstruktoru ve třídě není povinné.
- Konstruktor explicitně nevoláme, je volán automaticky při vzniku objektu.

Typické použití konstruktoru je inicializace členských proměnných objektu nebo alokace dynamických částí objektu.

```
struct Ctverec { float a;
                 Ctverec() { }                    // 1. konstruktor
                 Ctverec(float aa) { a=aa; }      // 2. konstruktor
                 float obsah() { return a*a; }
```

```
};
```

Má-li konstruktor parametry, jejich argumenty uvádíme při definování objektu.

```
Ctverec c1,c2(),c3(5);   // c1,c2 - 1. konstruktor, c3 - 2. konstruktor
```

Konstruktor s jedním parametrem je tzv. konvertující konstruktor (pokud není před ním uvedeno klíčové slovo *explicit*). Definici objektu můžeme zapsat pomocí operátoru přiřazení:

```
Ctverec c4=2;    // 2. konstruktor

struct Ctverec { float a;
                 Ctverec() { }
                 explicit Ctverec(float aa) { a=aa; }
                 float obsah() { return a*a; }
};
Ctverec c5=8;    // chyba !!  2. konstruktor není konvertující konstruktor
```

Pro inicializaci proměnných v jejich deklaraci lze vedle přiřazení počáteční hodnoty operátorem přiřazení rovněž použít *inicializátor*. Výraz, kterým je proměnná inicializována, je uzavřen v závorkách za proměnnou.

```
int i(3);    // int i=3;
```

Inicializátor lze použít v konstruktoru třídy pro přiřazení počáteční hodnoty proměnným třídy, kde za hlavičkou konstruktoru uvedeme znak **:** a za ním seznam inicializátorů (jednotlivé inicializátory jsou za sebou a vzájemně oddělené čárkami).

```
struct Obdelnik { float a,b;
                  Obdelnik(float a,float b): a(a),b(b) { }
};
```

Členské proměnné, která je referencí, nelze přiřadit počáteční hodnotu jinak než inicializátorem.

```
struct Bod { float x,y; };

struct Usecka { Bod &b1,&b2;
                Usecka(Bod &b1,Bod &b2): b1(b1),b2(b2) { }
};
```

Vlastnosti funkce destruktoru:

- Jeho jméno začíná znakem ~ a za ním je jméno třídy.
- Nemá žádné parametry.
- Nevrací hodnotu a ani se před jeho jménem neuvádí klíčové slovo *void* .
- Uvedení destruktoru ve třídě není povinné.
- Destruktor explicitně nelze zavolat, je volán automaticky při zániku objektu.

Typické použití destruktoru je pro úklid v objektu (uvolnění dynamicky alokované paměti v objektu, zavření souborů používaných v objektu apod.).

```
struct Pole { int n;
              float *p;
              Pole(int m) { p=new float [n=m]; }
              ~Pole() { delete [] p; }
};
```

Pro definici členské funkce platí:

- Může být uvnitř definice třídy – taková funkce má charakter *inline* funkce.
- Může být vně definice třídy, uvnitř třídy je jen její deklarace (prototyp). Definice uvedená vně třídy má před jménem funkce jméno třídy oddělené operátorem rozlišení **::** .

```
struct Pole { int n;
              float *p;
              Pole(int m) { p=new float [n=m]; }
              void tridit();
              ~Pole() { delete [] p; }
};

void Pole::tridit() { … }
```

Můžeme mít definici všech členských funkcích vně třídy. U těch, které mají mít charakter *inline* funkcí, uvedeme u jejich definice klíčové slovo *inline*.

```
struct Pole { int n;
              float *p;
              Pole(int);
              void tridit();
              ~Pole();
};

inline Pole::Pole(int m) { p=new float [n=m]; }

void Pole::tridit() { … }

inline Pole::~Pole() { delete p; }
```

Zapouzdření

Je základní rys objektově orientovaného programování. Má dva aspekty:

- Proměnné a funkce, které s nimi pracují, jsou spojeny v jeden celek – třídu.

- Členská proměnná třídy je chráněna proti tomu, aby její hodnotu nemohly změnit (a případně i používat) jiné funkce, než jsou funkce třídy, ve které je proměnná definována. Rovněž i členská funkce třídy může být chráněna proti tomu, aby je nemohla volat jiná funkce než členská funkce stejné třídy.

Omezení přístupu k členům třídy:

private	soukromý	přístupný jen pro funkce dané třídy
protected	chráněný	přístupný i pro funkce dědících tříd
public	veřejný	veřejně přístupný

Implicitní omezení přístupu k členům třídy:

struct	**public**
class	**private**

Konstruktory a destruktor musí být veřejné funkce (chceme-li vytvořit nějaký objekt třídy).

```
class Ctverec { float a;
        public: Ctverec(float a): a(a) { }
                float obsah() { return a*a; }
};
Ctverec c(4);
c.a=3;                  // chyba !!
cout << c.a;            // chyba !!
cout << c.obsah();      // lze
```

```
class Ctverec { float a;
        public: Ctverec(float a): a(a) { }
                float strana() { return a; }
                float obsah()  { return a*a; }
};
Ctverec c(4);
cout << c.strana();
cout << c.obsah();
```

```
class Ctverec { float a;
        public: Ctverec(float a): a(a) { }
```

```
                void zmenit(float aa) { a=aa; }
                float strana() { return a; }
                float obsah()  { return a*a; }
};
Ctverec c(4);
c.zmenit(3);
cout << c.strana();
cout << c.obsah();
```

Kopírující konstruktor

Vytváří nový objekt jako kopii již existujícího objektu. Je implicitně definován v každé třídě.

Zápis deklarace objektu s použitím kopírujícího konstruktoru:

```
  třída objekt2(objekt1);
  třída objekt2=objekt1;
```

```
class Bod { float x,y;
    public: Bod(float xx,float yy) { x=xx,y=yy; }
};
Bod b1(1,2);
Bod b2(b1);
Bod b3=b1;
```

Kopírující konstruktor dělá mělké kopírování. Ve většině případů je to postačující. Někdy je ale zapotřebí hluboké kopírování. Jsou to zejména případy, kdy jsou v objektu dynamicky vytvářené datové struktury.

```
struct Jmeno { char *r;
  Jmeno(const char *s) { r = new char [21]; strcpy(r,s); }
  void zmenit(const char *s) { strcpy(r,s); }
  ~Jmeno() { delete [] r; }
};
Jmeno e("Eva");
Jmeno j=e;
j.zmenit("Jana");
cout << e.r;    // Jana
```

```
cout << j.r;    // Jana
```

```
struct Jmeno { char *r;
  Jmeno(const char *s)  { r = new char [21]; strcpy(r,s); }
  Jmeno(const Jmeno &j) { r = new char [21]; strcpy(r,j.r); }
  void zmenit(const char *s) { strcpy(r,s); }
  ~Jmeno() { delete [] r; }
};

cout << e.r;    // Eva
cout << j.r;    // Jana
```