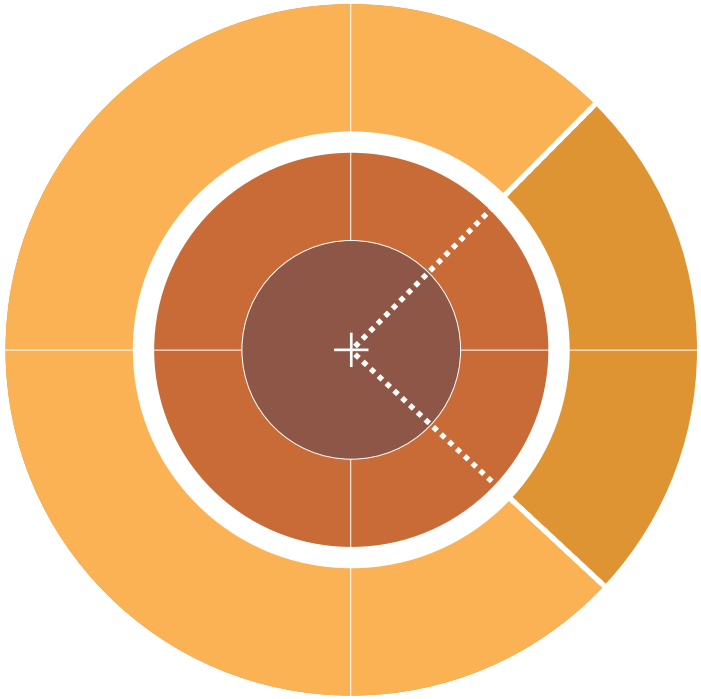
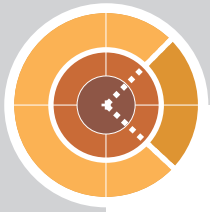


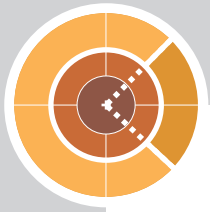


Objekty v C++

- **objekt je instancí třídy**
- **třída** definuje **nový datový typ**, uživatelský datový typ (protože ho vytváří uživatel-programátor)
- objekty mohou mít asociované proměnné, tzv. **členské proměnné**, a také asociované funkce, tzv. **metody**

```
class MyClass {        <-- definice třídy
  /* private, public a protected
     proměnné, konstanty a metody */
};

MyClass myclass;       <-- deklarace objektu
```

- třídy se pojmenovávají podle toho, co v aplikaci provádí nebo představují => pokud nemůžete vymyslet pro třídu vhodné jméno, máte **aplikaci špatně dekomponovanou**



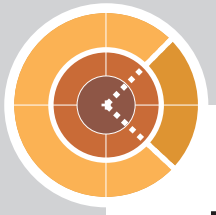
Definice třídy

```
class Point {
  double x, y;
 public:
  void setx(const double x);
  void sety(const double y);
};

void Point::setx(const double x) {
  this->x = x;
}

void Point::sety(const double y) {
  this->y = y;
}
```

členské proměnné (není-li uveden přístupový specifikátor, jedná se o **private**)

za definicí třídy **musí** být středník

ukazatel na instanci



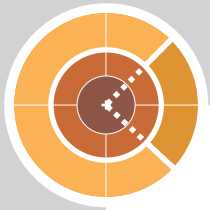
Deklarace objektu

```
#include <iostream>
…
int main() {
  Point p;

  p.setx(5.0);
  p.sety(7.0);
  cout << p.x;

  return 0;
}
```

deklarace je stejná, jako u jiných typů - ale pozor, objekt je v tomto případě **statický**

nelze! `x` je `private`

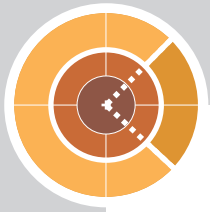
- před voláním funkcí `p.setx()` a `p.sety()` je hodnota `x` a `y` samozřejmě náhodná, **neinicializují se**...



Specifikátory přístupu k členům objektu

```
class Bottle {
  private:
    double level;
  protected:
    double volume;
  public:
    char brand[32];
    void setlevel(double level);
};
```

- **private** (i když není uvedeno nic): ke členu má přístup jen jiný člen **téže třídy** - např. metoda **setlevel()** může modifikovat **private level**, nikdo "zvenčí" třídy nemůže...
- **protected**: podobné **private** - viz dále dědičnost
- **public**: člen je veřejně přístupný, lze jej modifikovat odkud je libo (z jiné třídy, hlavního programu, apod.)



Členské funkce čili metody

```
class Bottle {
  double level;
  public:
    char brand[32];
    void setbrand(const char *brand) {
      strcpy(this->brand, brand); }
    void setlevel(double level);
};

void Bottle::setlevel(double level) {
  this->level = level;
}
```

- metoda může být definovaná v těle definice třídy nebo vně
- uvnitř se definují obvykle jen krátké metody (get-/settery)
- uvnitř musí být vždy deklarovaná, deklarace se nesmí lišit

Metody s modifikátorem const

```
class Bottle {
  double level;
  public:
    const double *getlevel() const;
};

const double *Bottle::getlevel() const {
  return &level;
}
```

- **const** říká překladači, že metoda nemodifikuje žádné členské proměnné třídy, tj. nemění stav objektu
- **const** metoda nemůže volat ne-**const** metodu (v případě, že se o to programátor pokusí, překladač si stěžuje)

Shrnutí použití metod s modifikátorem const

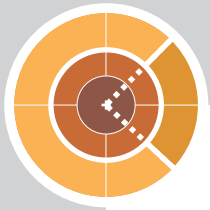
```
class GUI {
  public:

    Widget *widget();

    Widget *widget() const;

    const Widget *cWidget()

    const Widget *cWidget() const;

  private:
    Widget *m_widget;
};
```

modifikuje **m_widget** a uživatel může modifikovat vrácený **Widget**

nemodifikuje **m_widget**, ale uživatel může modifikovat vrácený **Widget**

modifikuje **m_widget** a uživatel nesmí modifikovat vrácený **Widget**

nemodifikuje **m_widget** a ani uživatel nemůže modifikovat vrácený **Widget**



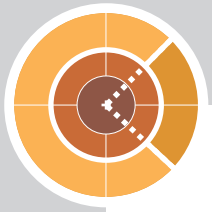
Konstruktor

- speciální metoda, jejímž úkolem je **objekt vytvořit**
- **jmenuje se vždy stejně jako třída, které náleží**
- překladač volá konstruktor vzápětí po alokaci "surové" paměti pro objekt - konstruktor paměť naplní, vyčistí, atp.

```
class Bottle {
  double level;
  public:
    Bottle() { level = 0.5; }
    double getlevel() { return level; }
};
```

při deklaraci statické instance objektu zajistí volání konstruktoru překladač

- konstruktor může být přetížený, **nemusí být vůbec uveden**
- pokud je deklarován, musí být i definován (uvnitř nebo vně)

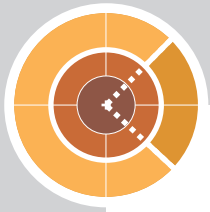


Konstruktor s inicializátorem proměnných

- součástí deklarace konstruktoru může být **inicializace** členských proměnných třídy (jedné nebo více oddělených čárkami)

```
class Bottle {
  double level, volume;
  public:
    Bottle():level(0.5), volume(0.75) { }
    double getlevel() { return level; }
};
```

tělo konstruktoru může být prázdné, pokud jeho jediným úkolem je inicializace proměnných



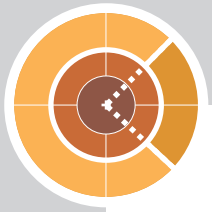
Přetížený konstruktor, konstruktor s default hodnotou

```
class Bottle {
  double level, volume;
  public:
  1 Bottle() {
      level = 0.75;
      volume = 0.75;
    }
  2 Bottle(double l, double v = 0.75) {
      level = l;
      volume = v;
    }
};
```

tato hodnota se použije, není-li parametr uveden

```
Bottle any;
Bottle wine(0.75);
Bottle beer(0.5, 0.5);
```

- hodnoty parametrů pro konstruktor se udávají v deklaraci do závorek za jméno proměnné (objektu)

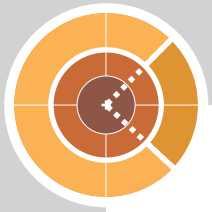


Zvláštní případ inicializace v konstruktoru

```
class Foo {
  int a, b, c;
  public:
    Foo(int, int, int);
};

Foo::Foo(int a, int b, int c): a(a),
  b(b), c(c) {
  /* kód konstruktoru */
}
```

- tato technika umožňuje oddělit logiku od inicializace proměnných, v deklaraci v těle stačí uvést typ (kvůli velikosti)

- **konstruktor nemá nikdy uvedený žádný návratový typ** (ani **`void`** - návratovým typem je sama instance objektu)

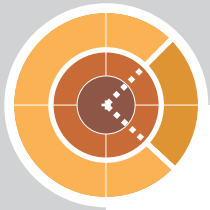


Přístup k private proměnným - tzv. konstruktor copy

```
class Foo {
  private:
    int val;
  public:
    Foo(const Foo &f) {
      val = f.val;
    }
};
```

v pořádku

- **různé instance téže třídy mohou přistupovat k `private` proměnným**
- toho využívají konstruktory **`copy()`**, které vytváří objekt jako kopii jiného objektu téže třídy



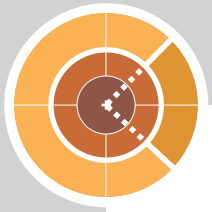
Destruktor

- speciální metoda, jejímž úkolem je **objekt zlikvidovat**
- musí uvolnit všechny prostředky (dynamické proměnné, apod.), které alokoval konstruktor
- **jmenuje se vždy stejně jako třída**, které náleží, a je uvozen znakem **~** (tilda)

```
class Matrix {
  double *data;
  public:
    Matrix() { data = new double[100]; }
    ~Matrix() { delete [] data; }
};
```

destruktor ruší dynamickou proměnnou vytvořenou konstruktorem

- volání destruktoru zajišťuje překladač před uvolněním paměti pomocí **`delete`** => **nelze použít na statické objekty**

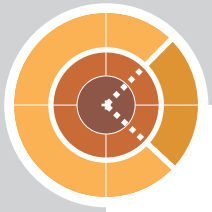


Vytváření dynamické instance objektu

```
class Test {
   int val;
   public:
      Test(): val(0) {}
      int getval() { return val; }
      void setval(int v) { val = v; }
};

int main() {
   Test *t;
   t = new Test();
   t->setval(5);
   delete t;
   return 0;
}
```

vytvoření dynamické instance objektu

zrušení dynamické instance objektu



Vytváření dynamické instance objektu

```
class Test {
  int val;
  public:
    Test(): val(0) {}
    Test(int v) { val = v; }
    int getval() { return val; }
    void setval(int v) { val = v; }
};

int main() {
  Test *t = new Test(7);
  t->setval(5);
  delete t;
  return 0;
}
```

vytvoření dynamické instance objektu

zrušení dynamické instance objektu

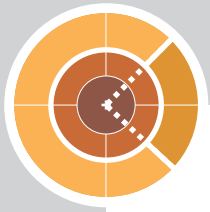
Přetěžování operátorů

```
class Test {
  int val;
  public:
    Test(): val(0) {}
    Test(int v) { val = v; }
    Test &operator =(const Test &right);
};

Test &Test::operator =(const Test &right) {
  if (this != &right) {
    this->val = right.val;
  }
  return (*this);
}
```

deklarace přetížení operátoru přiřazení instance objektu

umožňuje zápis `a = b = c;`

definice operátoru =
podmínka `if()` zajišťuje, že se objekt nepřiřadí "sám sobě" ...



Přetěžování operátorů (pokračování)

```
int main() {
   Test *t, *u;

   t = new Test(5);
   u = t;

   u->setval(7);
   cout << t->getval() << endl;
   cout << u->getval() << endl;

   delete t;
   delete u;

   return 0;
}
```

t->val == 5

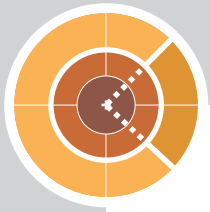
t->val == 7

objekt **u** je teď **aliasem t** v paměti existuje pouze jedna instance, odkazují na ní 2 proměnné...

přes **u** přistupujeme k **t**

tohle **není** volání našeho přetíženého operátoru (neodpovídá typ)

nemá smysl (**u** nebyl vytvořen) ale nezpůsobí to chybu



Přetěžování operátorů (pokračování)

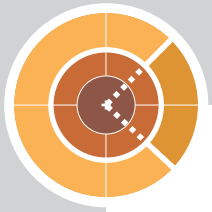
```
int main() {
   Test *t = new Test(5);
   Test *u = new Test(7);

   *u = *t;

   cout << t->getval() << endl;

   cout << u->getval() << endl;

   delete t;
   delete u;

   return 0;
}
```

u->val == 7

t->val == 5

u->val == 5

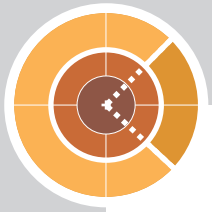
jedná se o 2 různé objekty samostatně existující v paměti
přiřazení => přetížený operátor,
vlastně **kopírování obsahu**



Přetěžování binárních operátorů (členskou funkcí)

```
class Vector {
  private:
    int len, items[];
  public:
    Vector(): len(0) {}
    Vector(int len) { this->len = len; }
    Vector &operator +(const Vector &rop);
};

Vector &Vector::operator +(
  const Vector &rop) {
  for (int i = 0; i < len; i++)
    this->items[i] += rop.items[i];
  return (*this);
}
```

volající objekt (**this**) je levým operandem



Přetěžování binárních operátorů (členskou funkcí) - použití

```
int main() {
   Vector *t = new Vector(3);
   Vector *u = new Vector(3);
   Vector *v = new Vector(3);

   *v = *t + *u;

   delete t;
   delete u;
   delete v;

   return 0;
}
```

dereference je **nutná**, jinak by se provádělo sčítání adres (hodnot ukazatelů)

```
int main() {
   Vector t(3), u(3), v(3);
   v = t + u;

   return 0;
}
```

ve statickém případě takto

Přetěžování binárních operátorů (mimo třídu)

binární operátor jako nečlenská funkce

```
Vector operator +(const Vector &leftop,
  const Vector &rightop) {...}
```

- v případě nečlenské funkce **se předává levý i pravý** operand

Přetěžování unárních operátorů

unární operátor jako nečlenská funkce

```
Vector operator -(const Vector &vect) {...}
```

unární operátor jako členská funkce

```
Vector Vector::operator -() {...}
```

- operandem se rozumí volající objekt (**this**)