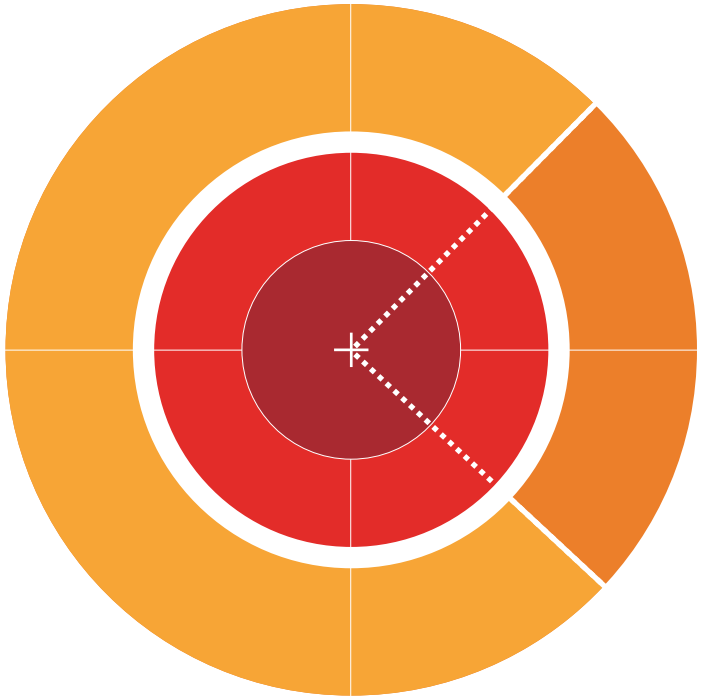


Dědičnost tříd

- **dědičnost** je vztah mezi dvěma **nebo více** třídami, přičemž jedna (nebo více) je **rodičem** (parent, base) a jedna je **potomkem** (child, subtype)
- **vícenásobná dědičnost** je situace (pouze v C++), kdy potomek dědí od více různých rodičů

```
class MyClass : ParentClass {
   ...
};
```

rodič

potomek

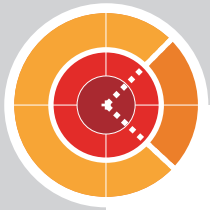
- dědičnost může být deklarovaná jako **`public`**, **`private`** nebo **`protected`**
- potomek má (všechny) proměnné a metody rodiče

Příklad dědění od jediného rodiče

```
class Form {
    int area;
  public:
    int colour;
    int getArea(){ return this->area }
    void setArea(int area) {
      this->area = area }
};

class Circle: public Form {
    int diameter;
  public:
    int getDiameter();
    void setDiameter(int diameter);
    bool isDark() { return colour > 10; }
};
```

jediný předek (ancestor)

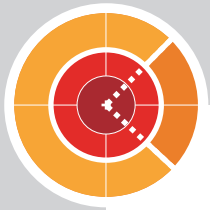
dědí se **veřejně**
(public inheritance)



Specifikace dědění - public, private, protected

dědičnost	členská proměnná nebo metoda: private	public	protected
public	private	public	protected
private	private	private	private
protected	private	protected	protected

- účelem je zvýšení bezpečnosti při používání knihoven, připravených jiným programátorem



Význam specifikátorů dědění

`public`

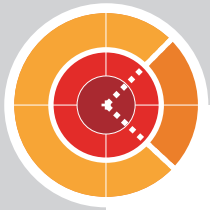
- **člen je přístupný odkudkoliv**, lze jej **číst** i **modifikovat**, do potomků se zdědí tak, že přístupnost v potomkovi je dána specifikací dědění...

`private`

- **člen je přístupný pouze uvnitř třídy**, metody třídy ho mohou číst i modifikovat - **zvenčí je neviditelný**; potomek ho zdědí, ale je pro něj neviditelný... musí s ním pracovat pomocí veřejných get-/setterů rodiče

`protected`

- **člen je přístupný pouze uvnitř třídy a jejích potomků**, jinak se chová stejně jako **`private`**



Specifikace dědění protected

- člen označený **`protected`** se při **`public`** dědičnosti dědí jako **`protected`** - ke členu nelze přistupovat mimo třídu a zároveň **potomci** takové třídy **mají k tomuto členu přístup**
- někdy výše uvedený postup **není vhodný**, protože rodičovská třída by se měla o své detaily postarat sama a nepropagovat je do potomků
- bude-li mít jiný programátor, vytvářející potomka od naší třídy, přístup ke všem manipulačním metodám a vnitřním stavovým proměnným, určitě provede něco, co nechceme a co vede k problémům

DOBRÝ POSTUP: Nepoužívat **`protected`**, pokud lze stejného výsledku dosáhnout s **`private`**

Polymorfismus

- nový koncept v objektově orientovaných jazycích
- **jednoduše řečeno**: polymorfismus umožňuje zacházet s členy (zejména metodami) potomka stejně jako se členy rodiče

```
objcontainer.add(new ChildClass1());
objcontainer.add(new ChildClass2());
objcontainer.add(new ChildClass3());

for (int i = 0; i < objcontainer.size; i++)
  ((ParentClass *) objcontainer.get(i))->show();
```

potomci třídy `ParentClass`

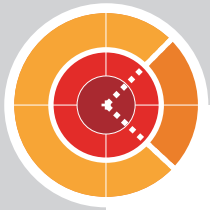
- může se jednat o naprosto různý výkonný kód, volání je však pořád stejné (protože se volá metoda předka), volaná metoda ovšem musí být v potomkovi definována, jinak se volá stejnojmenná metoda rodiče - to lze řídit specifikátorem `virtual`

Polymorfismus (příklad)

```
class Foo {
  public:
    void f() {
      cout << "Foo::f()" << endl;
    }
    virtual void g() {
      cout << "Foo::g()" << endl;
    }
};

class Bar : public Foo {
  public:
    void f() {
      cout << "Bar::f()" << endl;
    }
    virtual void g() {
      cout << "Bar::g()" << endl;
    }
};
```

`f()` není **`virtual`** a tedy volání způsobí provedení metody podle typu ukazatele

```
int main() {
  Foo foo;
  Bar bar;
  Foo *baz = &bar;
  Bar *quux = &bar;
  foo.f();    // "Foo::f()"
  foo.g();    // "Foo::g()"
  bar.f();    // "Bar::f()"
  bar.g();    // "Bar::g()"
  baz->f();   // "Foo::f()"
  baz->g();   // "Bar::g()"
  quux->f();  // "Bar::f()"
  quux->g();  // "Bar::g()"
  return 0;
}
```



Ryze virtuální metody

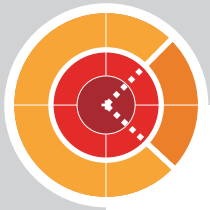
- **ryze virtuální** (nebo abstraktní) je taková metoda, jejíž definice v oděděné třídě (potomkovi) je vynucená, pokud není uvedena, překladač hlásí chybu

```
class Widget {
  public:
    virtual void paint() = 0;
};

class Button : public Widget {
  public:
    virtual void paint() { … }
};

Widget *w = window->activeWidget();
w->paint();
```

inicializátor ryze virtuální metody

v potomkovi je (musí být) definice této metody

- vlastně mechanismus **tvorby interface** => říkáme, že potomek musí mít nějakou funkci a implementace nás nezajímá



Vícenásobná dědičnost

- C++ umožňuje (jako jediný široce užívaný jazyk) vytvářet potomky od více rodičů současně (*multiple inheritance*)
- často to působí celou řadu problémů (**raději neužívat!**)
- v modernějších jazycích nahrazena technikou **`interface`**

```
class FlyingCat : public Cat, public FlyingAnimal {
  ...
};
```

- dědí se **členské proměnné** (tzv. atributy) a **metody** od obou rodičů => potomek má vlastnosti obou rodičů
- **POZOR:** to může způsobit dost závažnou kolizi

Kolize při vícenásobné dědičnosti