VÝNIMKY

Problém, ktorý sa snažíme riešiť

- Výskyt mimoriadnej udalosti, s ktorou sa nám nechce v kóde príliš zaoberať, pretože prakticky skoro nikdy nenastane a len by "znečisťovala" náš kód / algoritmus".
- Napríklad: vyčerpanie priestoru na disku, strata internetového spojenia, prechod zimný/letný čas, porucha meracieho senzora, ...

Riešenie bez použitia výnimiek

- Pridať k funkcii návratový kód, ktorý bude určovať, či funkcia skončila "normálne" (a výsledok je platný), alebo či nastala nejaká nečakaná situácia.
- Problém je, že ak nejaká funkcia má takýto návratový kód, tak potom pravdepodobne aj všetky funkcie, ktoré ju používajú budú mať takýto kód. Navyše po každom vyvolaní takejto funkcie, treba testovať hodnotu tohto kódu či náhodou nenastala takáto výnimočná situácia.
- Celkovo toto riešenie môže viesť k nečitateľnému programu, kde väčšina kódu sa zaoberá situáciami, ktoré pri behu skoro nenastávajú.

Riešenie C

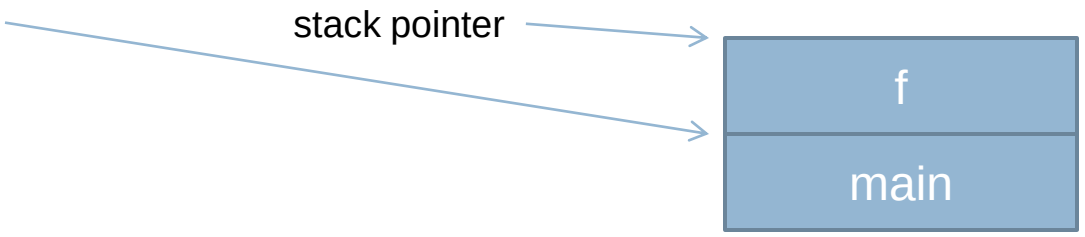
- Dvojica exotických funkcií v štandardnej knižnici, ktorá implementuje tzv. dlhý skok (long jump).
  - int setjmp(jmp_buf **`env`**) - do **`env`** uloží aktuálnu veľkosť systémového zásobníka (a zopár ďalších hodnôt určujúcich súčasný stav behu programu). Vráti návratovú hodnotu 0.
  - void longjmp(jmp_buf **`env`**, int val) - z **`env`** zrekonštruuje ukazateľ systémového zásobníka a behu programu ako bol v okamihu vyvolania **`setjmp`** a vráti sa z volania **`setjmp`** s návratovou hodnotou **`val`** (ktorá by mala byť nenulová).

Príklad

```
static jmp_buf env;

int f() {
  printf("f:1\n");
  longjmp(env, 33);
}

int main() {
  int r;
  r = setjmp(env);
  printf("main:1 r == %d\n", r);
  if (r == 0) {
       printf("main:2\n");
       f();
       printf("main:3\n");
  } else {
       printf("main:4\n");
  }
}
```



Vypíše:

```
main:1 r == 0
main:2
f:1
main:1 r == 33
main:4
```

Nikdy nevypíše

Riešenie C++

- Výnimky. Je to mechanizmus priamo v jazyku C++. Ide o akýsi vylepšený / zovšeobecnený setjmp - longjmp.
- Zaviedli sa kľúčové slová **`try`**, **`catch`** a **`throw`**. **`Try`** určuje časť kódu, odkiaľ sa "odchytia" výnimky. **`Catch`** určuje typ výnimky na "odchytenie" a kód, ktorý sa vzkoná v prípade výnimky. **`Throw`** sa použije v prípade, že nastala výnimka a riadenie sa má preniesť na príslušný catch.

```
try {
   kód
} catch (Typ premenná) {
   kód
}
```

Môže sa opakovať

```
throw výraz;
```

Výnimky C++ príklad

```
int f() {
  printf("f:1\n");
  throw 77;
}

int main() {
  try {
    printf("main:2\n");
    f();
    printf("main:3\n");
  } catch (int e) {
    printf("main:4 e == %d\n", e);
  }
}
```

Vypíše:

```
main:2
f:1
main:4 e == 77
```

Catch all

```
int fun() {
  try {
    nejaký kód
  } catch (Type1 var1) {
    kód pre výnimku typu Type1
  } catch (Type2 var2) {
    kód pre výnimku typu Type2
  } catch (...) {
    kód pre všetky ostatné výnimky
  }
}
```

Deklarácia výnimiek

```
int f() throw (SomeClass) {
   ...
      throw new SomeClass();
   ...
}
```

Deklarácia výnimiek je v C++ nepovinná!

Výnimky na rozdiel od longjmp vyvolávajú deštruktory

```
class A {
public:
  ~A() {printf("~A()"); }
};

int f() {
  throw 77;
}

int fun() {
  A a;
  f();
}

int main() {
  try { fun(); } catch(...) {}
}
```

ZMENA TYPU (CASTING)

Prečo potrebujeme meniť typ

- OOP vyžaduje zmenu typu v rámci hierarchie na implementáciu kontainerov.
- malloc vracia void *, treba ho konvertovať na X*.
- Pre zvýšenie smerníka o nejaké číslo, treba často konvertovať na char* a potom späť.
- Zmena typu môže vyjadrovať zmenu rozsahu premennej (výrazu). Napr. int -> long, int -> double, ...
- Občas proste treba niečo zahackovať, aby to chodilo.

Použitie kontainera v OOP vyžaduje zmenu typu

```
class Object { };

class Queue {
public:
  int size();
  void put(Object *e) ;
  Object *get() ;
};
```

```
class Complex : public Object {
  double x, y;
public:
  void add(Complex *x);
};

void addToQ(Queue *s, Complex *o) {
  int i;
  Complex *c;
  for(i=0; i<s.size(); i++) {
      c = (Complex *) s->get();
      c->add(o);
      s->put(c);
  }
}
```

Zmena typu Object*->Complex*

Žiadna zmena typu na tomto mieste, konverzia Complex*->Object* je automatická

Zmena typu v jazyku C

- Konštrukcia: **x =(*typ*) y;**

- Umožňuje meniť akýkoľvek celočíselný typ (včítane smerníkov) na akýkoľvek iný celočíselný typ. Rovnako umožňuje meniť reálny a celočíselný typ. Napr. Smerník na A na smerník na B, int na long, double na int, unsigned na smerník na C, ...

Zmena typu v C++

- Chceme niečo rafinovanejšie ako C.
- Chceme aj takú konverziu (mimo iných), aby spätná zmena typu objektu (v rámci hierarchie dedenia) na pôvodný typ prebehla bez chyby, ale iná zmena by vyhodila v run-time chybu.
- Samotná statická informácia získaná počas kompilácie na to nestačí, potrebujeme informáciu o type objektu vo vnútri objektu. Potrebujeme RTTI (run time type information)

Štandardná trieda type_info

```
class type_info {
public:
  virtual ~type_info();
  bool operator ==(const type_info &x) const;
  bool operator !=(const type_info &x) const;
  bool before(const type_info &x) const;
  const char *name() const;
private:
  type_info(const type_info &x);
  type_info &operator=(const type_info &x);
};
```

Štandardný operátor typeid(...)

- Štandardný operátor **`typeid`**(...) sa používa podobne ako perátor **`sizeof`**(...), t.j. dá sa aplikovať na deklaráciu typu, alebo nejaký výraz. V oboch prípadoch je jeho výsledkom objekt typu **`type_info`**.

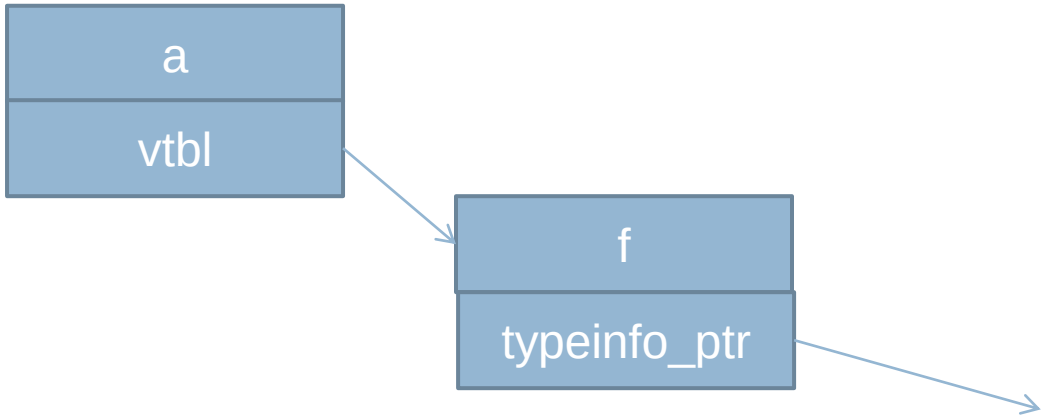
Napr:

```
typeid(int).name      → "i"
typeid(int *).name    → "Pi"

class MojaTrieda *p;
typeid(p).name        → "P10MojaTrieda"
```

Kde v pamäti sa type_info nachádza

```
class A {
int a;
virtual void f();
...
};
```



Type_info sa nachádza v tabulke virtuálnych metód. Dynamická informácia o type sa dá teda získať iba pre triedy obsahujúce aspoň jednu virtuálnu fukncíu.

dynamic_cast

- typeid je síce užitočná vec, ale v skutočnosti potrebujeme častejšie ako presný typ objektu testovať, či je podtriedou nejakej triedy v hierarchii. Niečo podobné ako je v Jave konštrukcia **`x.instanceof`**(*`Trieda`*).
- V C++ je to vyriešené zaradením bezpečnej konverzie v rámci hierarchie typov, t.j. konštrukciou **`dynamic_cast`**`<`*`Trieda`*` *`**`>(x)`**
- dynamic_cast otestuje, či v premennej **`x`** je v skutočnosti uložený smerník na objekt typu *`Trieda`*, alebo niektorú z jej podtried. Ak áno, výsledkom je smerník na objekt typu *`Trieda`*, inak NULL.
- Ale dynamic_cast potrebuje RTTI. Je preto ho možné použiť len na triedy s virtualnou funkciou.

Použitie kontainera s dynamic_cast

```
class Object { };

class Queue {
public:
 int size();
 void put(Object *e);
 Object *get() ;
};
```

Zmena typu Object*->Complex*

```
class Complex : public Object {
  double x, y;
public:
  void add(Complex *x);
};

void addToQ(Queue *s, Complex *o) {
  int i;
  Complex *c;
  for(i=0; i<s.size(); i++) {
      c = dynamic_cast<Complex *>(s->get());
      if (c == NULL) {
          ...
      } else {
        c->add(o);
        s->put(c);
      }
  }
}
```

Virtual base class and casting

```
class A {
  int a;
  virtual ~A() {}
};

class B : public virtual A {
  ...
};

void fun(A *pa) {
  B *pb;
  pb  = (B *) pa;
}
```

Chyba! Podľa štandardu C++
nemožno konvertovať pointer
z triedy získanej cez virtuálne dedenie
(Reprezentácia v pamäti vo všeobecnosti
neobsahuje back pointer z A na B)

Ale

```
class A {
  int a;
  virtual ~A() {}
};

class B : public virtual A {
  ...
};

void fun(A *pa) {
  B *pb;
  pb  = dynamic_cast<B *>(pa);
}
```

Ďalšie zjemňovanie konverzií

- Dynamic_cast je dobrý, ale zpomaľuje beh programu. Niekedy si je programátor na 100% istý, že konverzia je bezpečná a dá sa vzpočítať v čase kompilácie. V tom prípade môže použiť konštrukciu **`static_cast<`***`Trieda`***`*>(x)`**
- Static_cast je fajn, ale umožňuje len konverziu v rámci hierarchie dedení. Niekedy potrebujem skonvertovať takmer čokoľvek na čokoľvek. Pre takúto situáciu bol pridaný **`reinterpret_cast<`***`Typ`***`>(x)`**

Const_cast

- Existuje ešte **const_cast<*Typ*>(x)**, ktorý umožňuje pridávať / odoberať z typu modifikátor **const**. Používa sa teda na konverzie typu const char* -> char *, ...

```
void f(char *x) ;

int main() {
  f(const_cast<char *>( "hello" ));
}
```

Sumár: C++ má 5 typov konverzií

- `dynamic_cast<`*Trieda* `*>(x)`
- `static_cast<`*Typ*`>(x)`
- `reinterpret_cast<`*Typ*`>(x)`
- `const_cast<`*Typ*`>(x)`
- `(`*Typ*`)x`