Undantag

Varför använda undantag? För att alternativen ofta är sämre.

När ska man använda undantag? När något är/går fel och man inte vet hur man rättar till det.

Alternativ till undantag:
1. avsluta programmet
2. returnera ett värde som signalerar att fel har uppstått
3. försöka att fortsätta programkörningen ändå

```
struct RiddlesException {
  RiddlesException(std::string s) : description(s) {}
  std::string description;
};

std::map riddles;
std::string answer_to_riddle(std::string riddle) {
  if(riddles.find(riddle) == riddles.end()) {
    if(riddles.size() == 0)
      throw RiddlesException("no riddles!");
    else
      throw RiddlesException("too hard");
  }
  return riddles[riddle];
}
```

```
int main() {
  std::cout << "Ask me!";
  std::string s;
  while(std::cin) {
    std::getline(std::cin, s);
    try {
      s = answer_to_riddle(s);
      int local_dummy = s.length();
      std::cout << "Easy, " << s << std::endl;
    } catch (RiddlesException) {
      std::cout << local_dummy; // error, local_dummy undefined
      std::cout << "I'm tired of this game now...\n";
      return 0;
    }
    std::cout << "Another!";
  }
  return 0;
}
```

Man kan kasta vad som helst:

```
class A {};
class B : public A {};

throw 2;
throw "hello?";
throw B();
throw std::logic_error("logic error occured");
```

Man kan fånga olika saker:

```
try { /* some stuff */ }
catch(int &i) { std::cerr << i;}
catch(char *s) { std::cerr << s;}
catch(A &a) { std::cerr << "A or B";}
catch(B &b) { std::cerr << "will never happen";}
catch(...) { std::cerr << "caught something";}
```

Man kan tala om vad man möjligen kan kasta:

```
int throws_some_things(int i) throw(std::exception)
{ /* do stuff */}
int never_throws(int i) throw() { /* do stuff */}
int throws_everything(int i) { /* do stuff */}
```

Virtuella funktioner får ej överlagras med nya saker att kasta:

```
struct A {
  virtual void x() throw(X);
  virtual void xy() throw(X, Y);
};
struct B : public A {
  void x() throw(X, Y); // error
  void xy() throw(X);   // ok
};
```

Man kan kasta nya exceptions i sin exception handler:

```
try { throw 2; }
catch(int i) {
  throw 5;
}
```

Man kan också kasta vidare det man fångat:

```
try { throw std::runtime_error("error again...");}
catch(...) {
  /* delete all allocated memory */
  throw; // kasta på nytt
}
```

Det går bra att fånga saker i initieringslistan i en konstruktor också:

```
struct E {
  E() { throw 3;}
};
struct A {
  int a;
  E e;
  A(int aa);
};
A::A(int aa) try : a(aa) { // initializing of e throws here
  /* constructor body */
} catch (int) {
  /* exception handler */
}
```

Oväntade undantag.

Om ett exception som man inte deklarerat att man kan kasta ändå kastas kommer programmet normalt avslutas. Det som händer är att unexpected() anropas som i sin tur anropar terminate().

Det finns sätt att ändra detta beteende.

```
int f(int i) throws(int, std::bad_exeption) {
  throw "hello"; // kastar något oväntat, leder till bad_exception
}
```

Man kan också ändra hur unexpected() fungerar genom att tala om vilken funktion den ska köra (istället för terminate()) på följande sätt:

```
void my_unexpected_handler();
std::set_unexpected(my_unexpected_handler);
```

Strömmar

cout, utmatning från programmet
cin, inmatning till programmet
cerr, utmatning av felmeddelanden och liknande
filer
strängar

Använder shift-operatorerna "«" och "»".

Två varianter, en vanlig funktion eller en medlemsfunktion hos strömmen

```
std::ostream &std::ostream::operator<<(const Object &);
std::ostream &operator<<(const std::ostream &, const Object&);
```

Man kan skriva ut/läsa in flera saker på en gång eftersom operatorerna returnerar strömmen. Tänk även på prioriteten hos operatorerna:

```
std::cout << 1+2
          << (1 << 2) // dvs 4
          << (1 > 2 ? 1 : 2) // ? : har låg prioritet
          << '\n';
```

Strömmar är ofta buffrade, flush (och ofta endl) gör så att bufferten tömms.

```
std::cout << "Important" << std::flush;
while(1);
```

Man kan också koppla en ström och en utström, då flushas utströmmen före varje operation på den första strömmen. cin och cout är kopplade så.

```
std::cin.tie(&std::cout);
```

Man kan läsa till/från filer med strömmar.

std::ifstream, läsa från fil
std::ofstream, skriva till fil
std::fstream, läsa och skriva

```
#include <fstream>
std::ifstream in("filename.txt");
char s[100];
in.getline(s,100); // dåligt
std::string l;
std::getline(in, l); // bra
int i;
in >> i;
in.close();
std::ofstream out("utfil.txt");
out << s << i << std::endl << l << std::endl;
out.close();
```

Man kan läsa och skriva i en std::string med strömmar

istringstream, läsa från sträng
ostringstream, skriva till sträng
stringstream, läsa och skriva

Det finns även istrstream, ostrstream och strstream som använder char * istället för std::string. Använd inte dem.

```
#include <sstream>
std::string line; std::getline(std::cin, line);
std::ostringstream out;
std::istringstream in(line);
for(int i = 0; in; ++i) {
  char temp; in >> temp;
  out << i << ": " << temp << std::endl;
}
std::cout << out.str();
```

Ibland vill man spara objekt på fil. Det kan då vara trevligt om objekten kan skriva sig själva (och senare läsa tillbaka sina värden) från en ström.

```
struct A {
  int size; int *data;
  A(int s) : size(s) { data = new int[size]; }
  void serialize(std::ostream &);
  void deserialize(std::istream &);
};
struct B {
  int size; int *data;
  B(int s) : size(s) { data = new int[size]; }
  void serialize(std::ofstream &);
  void deserialize(std::ifstream &);
};
```

```
void A::serialize(std::ostream &out) {
  out << size;
  for(int i = 0; i < size; ++i)
    out << ' ' << data[i];
  out << std::endl;
}
void A::deserialize(std::istream &in) {
  int s;
  in >> s;
  if(s != size) {
    delete [] data;
    size = s;
    data = new int[size];
  }
  for(int i = 0; i < size; ++i)
    in >> data[i];
}
```

```
void B::serialize(std::ofstream &out) {
  out.write(reinterpret_cast<char *>(&size),
            sizeof(size));
  out.write(reinterpret_cast<char *>(data),
            size * sizeof(*data));
}
void B::deserialize(std::ifstream &in) {
  delete [] data;
  in.read(reinterpret_cast<char *>(&size),
          sizeof(size));
  data = new int[size];
  in.read(reinterpret_cast<char *>(data),
          size * sizeof(*data));
}
```

Vi kan inte dumpa hela objektet på en gång eftersom vi då skulle skriva ned pekarvärdet för "data" istället för alla datavärden. Har man inga pekare eller liknande går det bra.

```
int main() {
  A a1(3), a2(10);
  a1.data[0] = a1.data[1] = a1.data[2] = 7;

  std::ostringstream os;  a1.serialize(os);
  std::istringstream is(os.str());  a2.deserialize(is);
  // ett jobbigare sätt att göra a2 = a1 ...

  B b1(2), b2(3);
  b1.data[0] = 1; b1.data[1] = 0;
  std::ofstream of("bfile", std::ios::out
                          | std::ios::trunc
                          | std::ios::binary);
  b1.serialize(of); of.close();
  std::ifstream ifs("bfile", std::ios::in | std::ios::binary);
  b2.deserialize(ifs); ifs.close();
  return 0;
}
```

Övriga tips

```
template <int I> class A{ int f() {return I*I;}};

template <> class A<0>{ int f();}; // hela klassen specialiserad
int A<0>::f() { return 0;} // OBS! inget template<>

template<> int A<1>::f() {return 1;}
// endast funktionen specialiserad, template<>

template<class T> class A {
  friend std::ostream &operator<< <>(const std::ostream&,const A&);
};

#include <iostream> // inte <iostream.h> !!
g++ -Wall
Skriv/kopiera en Makefile !
```