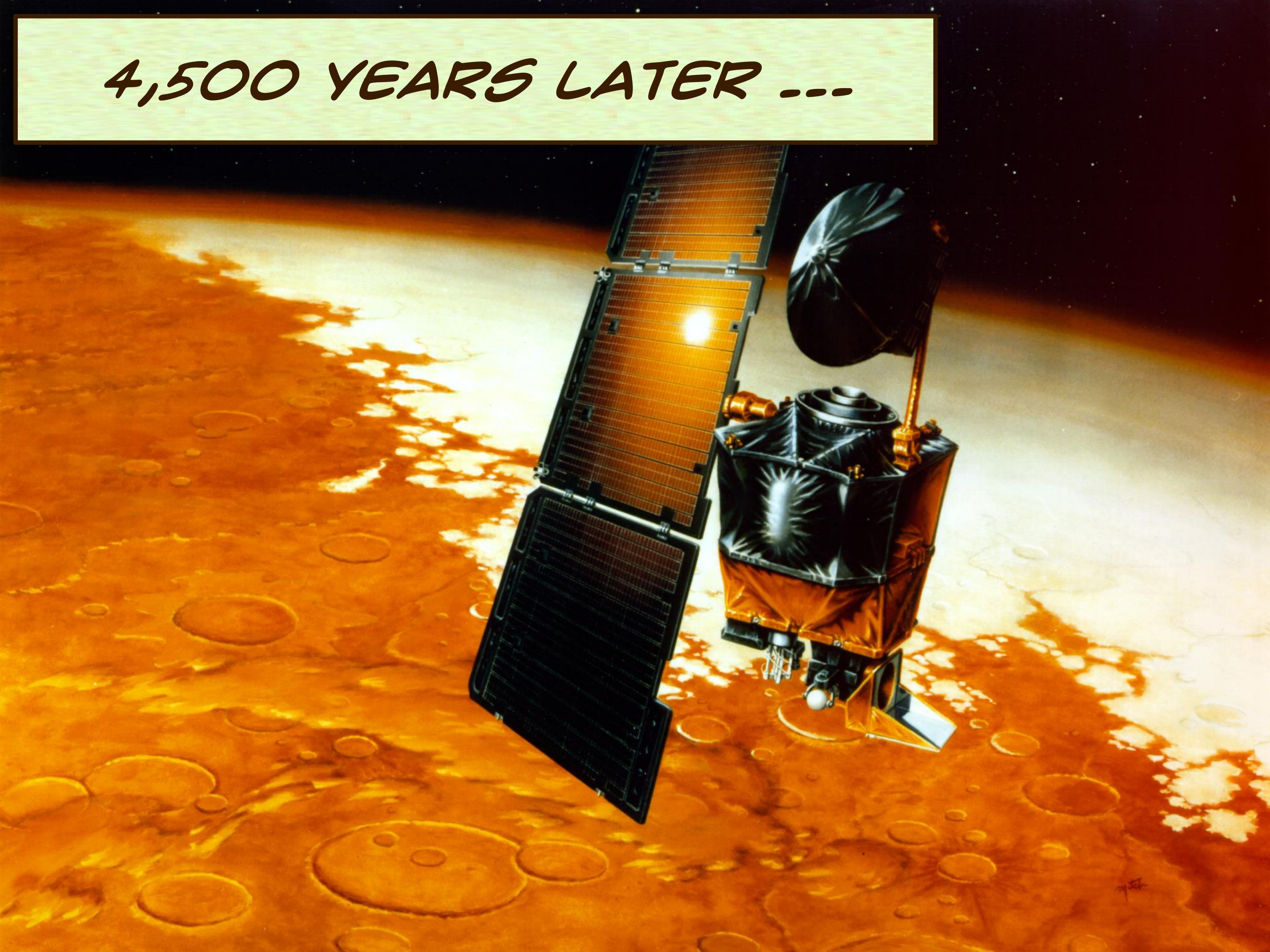
BUT LATER, MAN'S FREEDOM WAS TAKEN AWAY FOR THE 'GOOD OF THE STATE'...
MERCY, MASTER-- MERCY!
THERE IS NO MERCY FOR A SLAVE OF THE STATE! GET TO WORK--AND LET THESE PYRAMIDS BE A MONUMENT TO YOUR MISERY!

PUT YOUR TYPES TO WORK !!

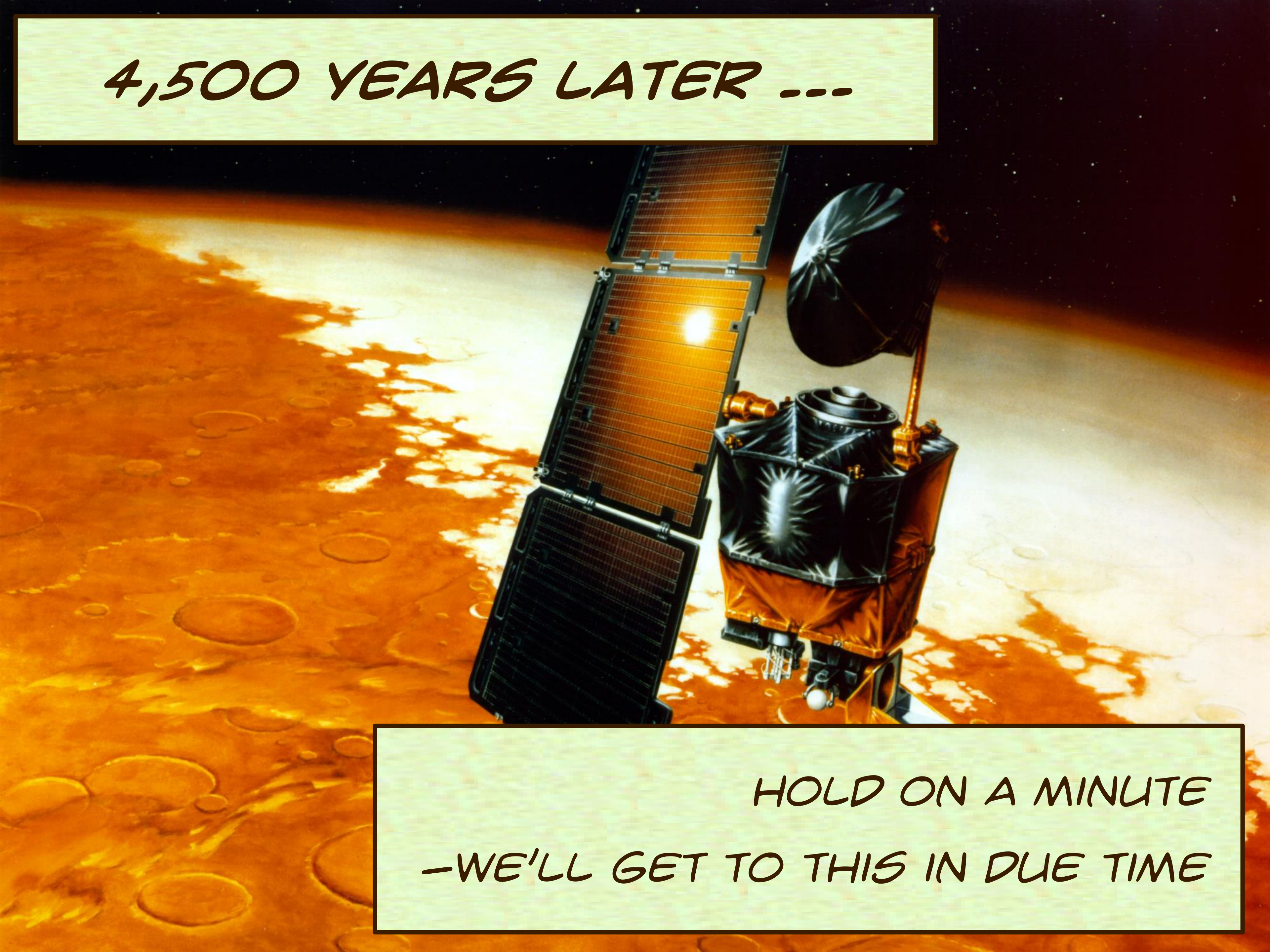
`using std::cpp 2016`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>
Madrid, November 2016

4,500 YEARS LATER ...



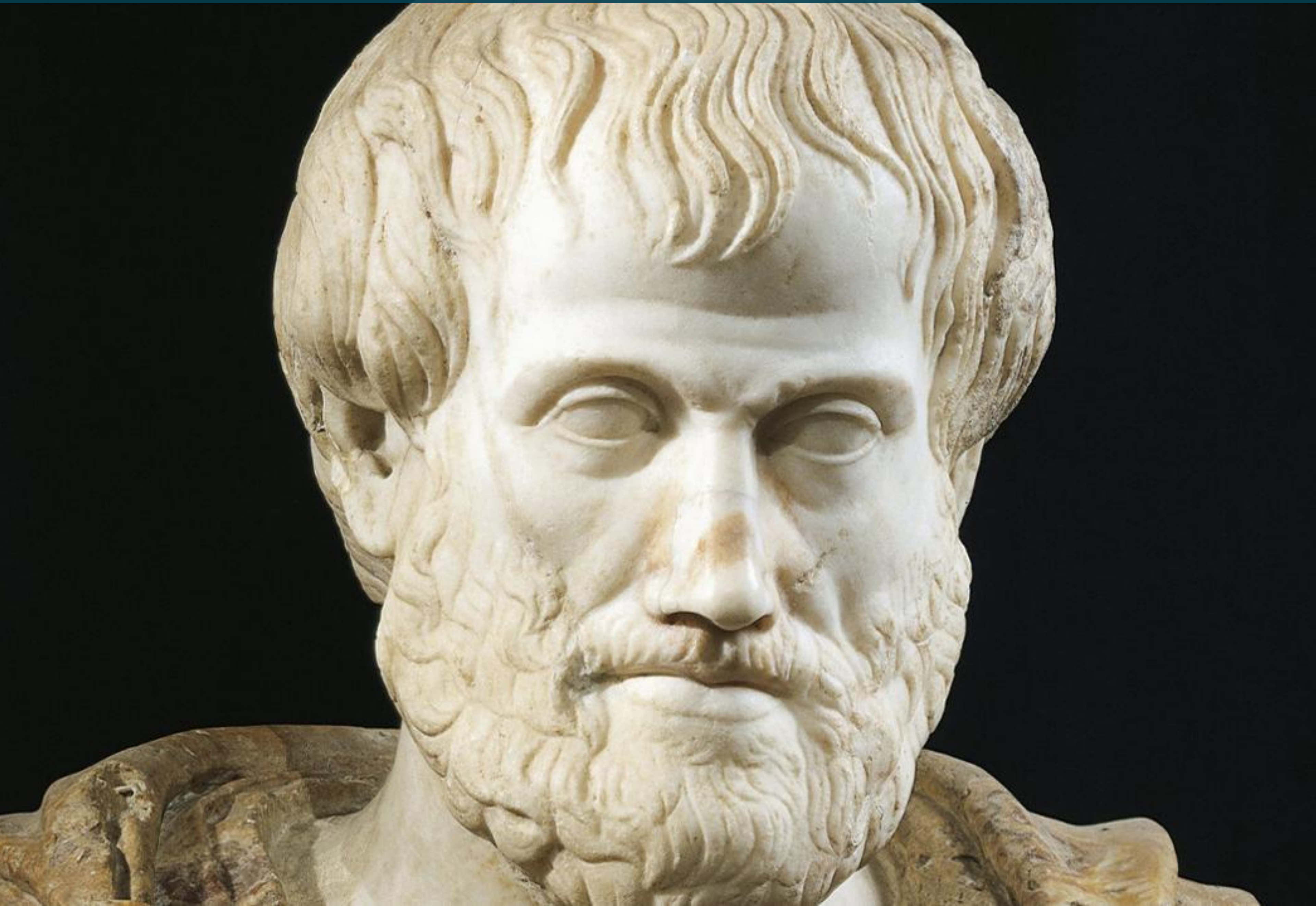
4,500 YEARS LATER ...



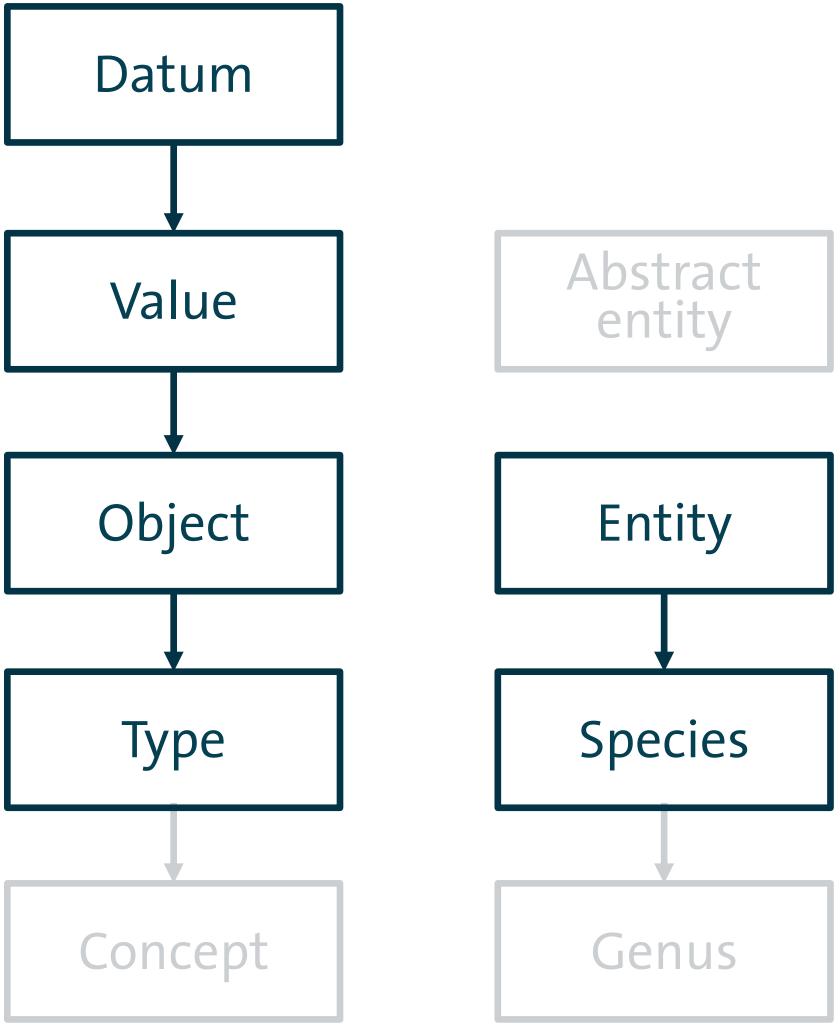
HOLD ON A MINUTE

—WE'LL GET TO THIS IN DUE TIME

What is a type, anyway?



What is a type, anyway?

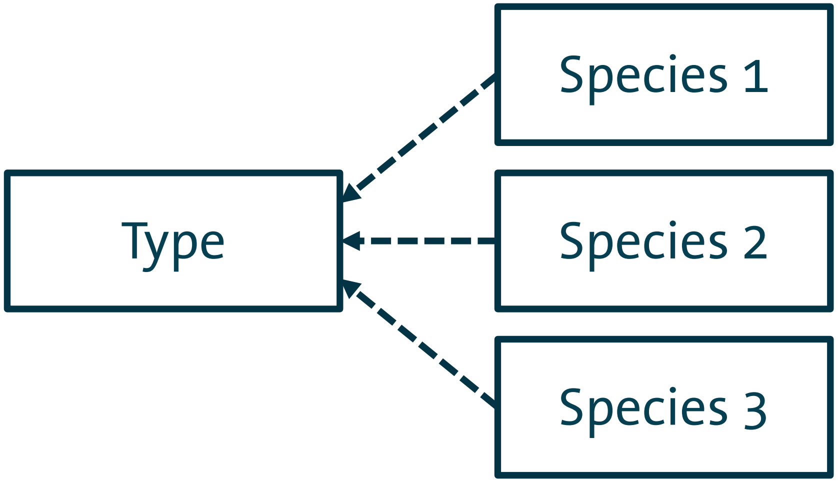


Alexander Stepanov
Paul McJones
Elements of programming
Addison-Wesley, 2009

What is a type, anyway?

- Programming is about modelling the real world
- Your types stand for real-world species, objects for entities
  - From math → app domain as we move higher up the hierarchy
- The state of an object is its value at a given (program) time
- Type invariants = permissible values
- Statically typed languages reduce permissible states of the world
  - The more info you give the compiler, the more bugs turn into compile-time errors

Bad practice #1: type overloading



Bad practice #1: type overloading

```
void set_date(int month,int day,int year)
{
  static const char* month_names[]=
  {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Dec"};

  std::cout<<month_names[month-1]<<" "<<day<<", "<<year<<"\n";
}

int main()
{
  // man lands on Moon
  set_date(20,7,1969); // oops
}
```

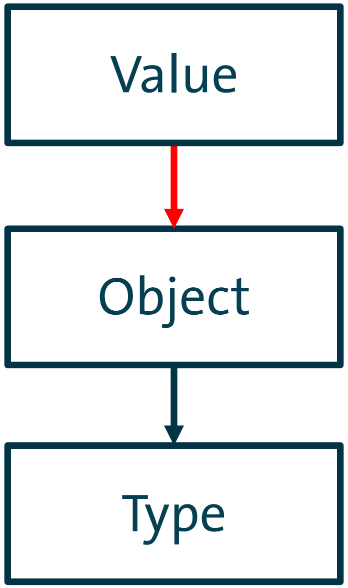
- A day (a month, a year) is **not** an `int`
- We are **representing** it with an `int`

Fighting type overloading

- At the very least, make types and representations different

```
struct month{int value;};
struct day{int value;};
struct year{int value;};

void set_date(month m,day d,year y)
{
  static const char* month_names[]=
  {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Dec"};

  std::cout<<month_names[m.value-1]<<" "<<d.value<<", "<<y.value<<"\n";
}

int main()
{
  // man lands on Moon
  set_date(day{20},month{7},year{1969}); // compile-time error
}
```

Bad practice #2: unchecked invariants



Bad practice #2: unchecked invariants

```
int main()
{
  // Apr 12, 1961: first spaceflight by Gagarin
  set_date(month{4},day{112},year{1961}); // oops
}
```

- Not all `int` values represent a valid day
- Except in the most trivial cases, object states are a **strict** subset of the representation type(s) values

Restricted representations

```
template<
  typename Int,
  Int min=std::numeric_limits<Int>::min(),
  Int max=std::numeric_limits<Int>::max()
>
class range{
  Int value;
  void check()const{assert(value>=min&&value<=max);}
public:
  range(Int value):value{value}{check();}
  range& operator=(Int value){this->value=value;check();return *this;}
  operator Int()const{return value;}
};
```

Restricted representations

```
struct month:range<int,1,12>{using range::range;};
struct day:range<int,1,31>{using range::range;};
struct year:range<int>{using range::range;};

void set_date(month m,day d,year y)
{
  static const char* month_names[]=
  {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Dec"};

  std::cout<<month_names[m]<<" "<<d<<", "<<y<<"\n";
}

int main()
{
  // Apr 12, 1961: first spaceflight by Gagarin
  set_date(month{4},day{112},year{1961}); // asserts
}
```

- Why not the following?

```
using month=range<int,1,12>;
```

Back to bad practice #1

```
int main()
{
  // March 31, 1966: Luna 10 probe is launched
  set_date(month{4},day{31},year{1966}); // oops
}
```

- Again, a problem with identifying objects and representations
- A date is **not** a tuple of (month, day, year)
- Keep iterating
- Rely on the experts…

Boost.Date_Time

```cpp
#include <boost/date_time/gregorian/gregorian.hpp>

using namespace boost::gregorian;

void set_date(date d)
{
  std::cout<<d<<"\n";
}

int main()
{
  // March 31, 1966: Luna 10 probe is launched
  set_date({greg_year{1966},greg_month{4},greg_day{31}}); // throws
}
```

Levels of invariant checking

- Level 0: no checking
- Level 1: choose a stricter representation/interface
- Level 2: checking on construction/assignment
- Level 3: full check

- Go at least for level 1-2

Level 1: improve this

```
class child
{
  parent* p;
public:
  child(parent* p):p{p}{}
};

void print(const char* msg);

void set_thermostat(int temperature);

const record* locate_record(int id);

struct elevator
{
  bool door_closed;
  int  speed;
};
```

Ideas from:
fonathan::blog(): Type safe - Zero overhead utilities for more type safety
http://tinyurl.com/gp34bu2
Ben Deane: Using Types Effectively
http://tinyurl.com/hzpzcdk

Possible alternatives

```cpp
class child
{
  parent* p;
public:
  child(parent& p):p{&p}{}
};

void print(const std::string& msg);

void set_thermostat(unsigned int temperature);

std::optional<std::reference_wrapper<record>> locate_record(int id);

struct elevator
{
  struct stopped{
    bool door_closed;
  };
  struct in_transit{
    int speed;
  };
  std::variant<stopped,in_transit> state;
};
```

Breaking the invariant barrier

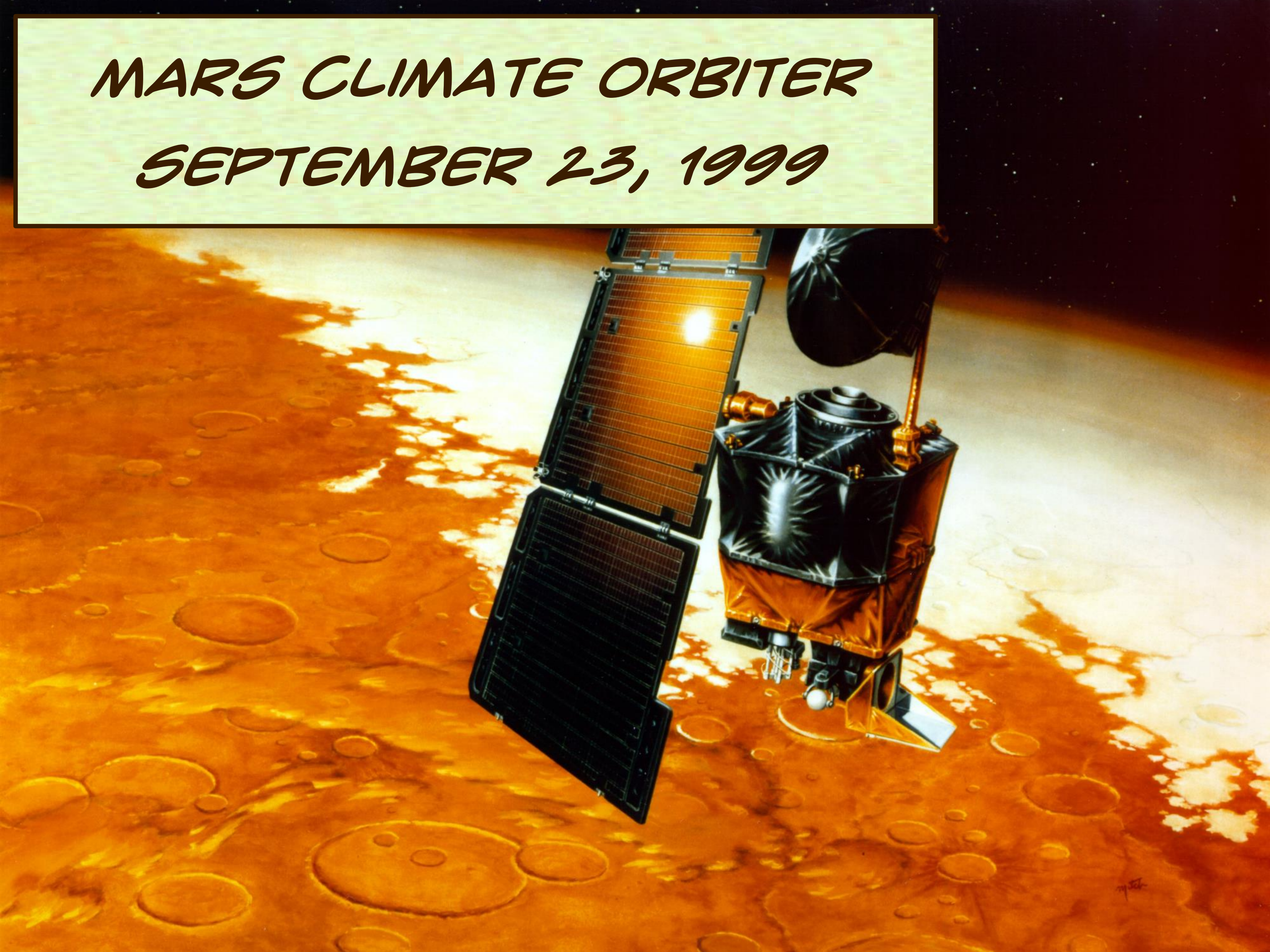
```
template<typename T>
class sorted_vector{
  using implementation=std::vector<T>;
  implementation impl;
public:
  using iterator=typename implementation::iterator;
  iterator begin(){return impl.begin();}
  iterator end(){return impl.end();}
  void insert(const T& x){impl.push_back(x);std::sort(begin(),end());}
  void erase(iterator it){impl.erase(it);}
};

int main()
{
  sorted_vector<int> sv;
  for(int i=0;i<10;++i)sv.insert(i);
}
```

- Now duplicate each element in `sv`

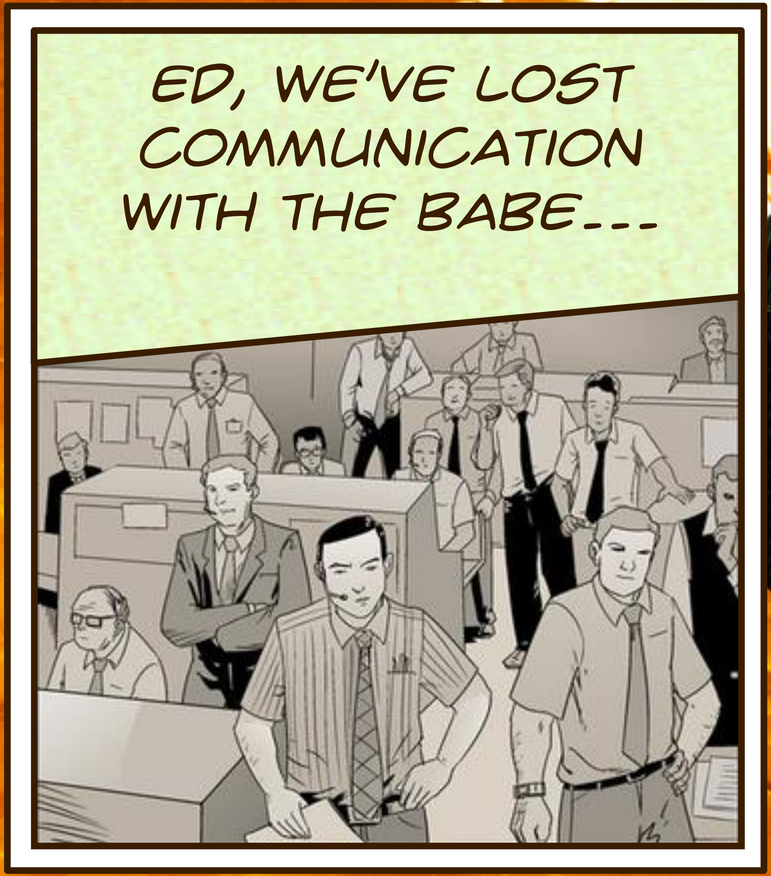
Breaking the invariant barrier

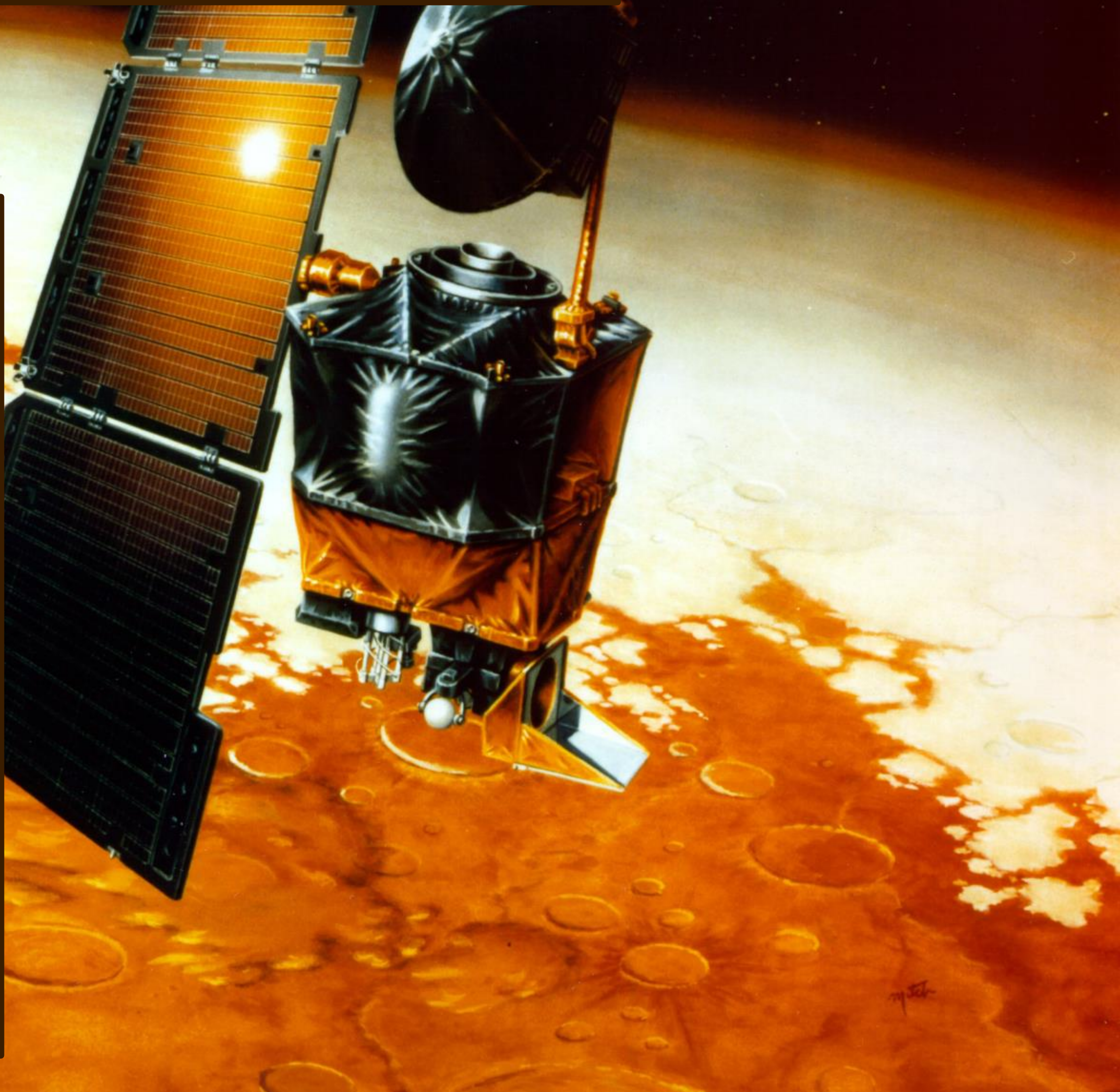
```cpp
template<typename T>
class sorted_vector{
  using implementation=std::vector<T>;
  implementation impl;
public:
  ...
  implementation extract(){return std::move(impl);}
  void accept(implementation&& i){
    impl=std::move(i);std::sort(begin(),end());
  }
};

int main()
{
  sorted_vector<int> sv;
  for(int i=0;i<10;++i)sv.insert(i);
  auto impl=sv.extract();
  for(std::size_t n=0,m=impl.size();n<m;++n)impl.push_back(impl[n]);
  sv.accept(std::move(impl));
}
```

MARS CLIMATE ORBITER
SEPTEMBER 23, 1999

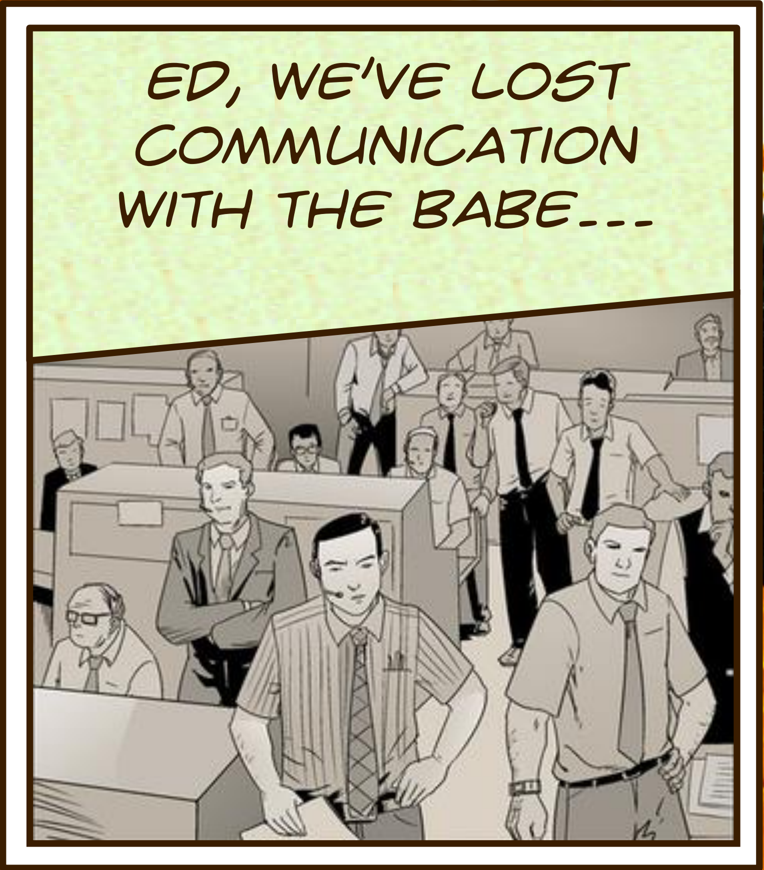


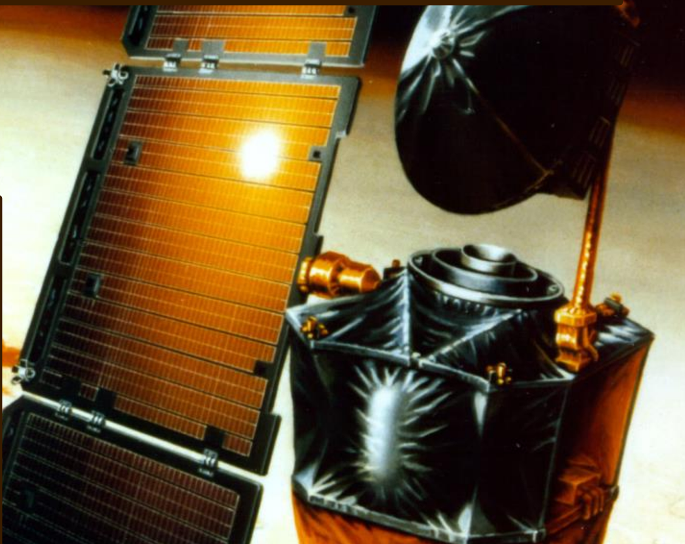
MARS CLIMATE ORBITER
SEPTEMBER 23, 1999





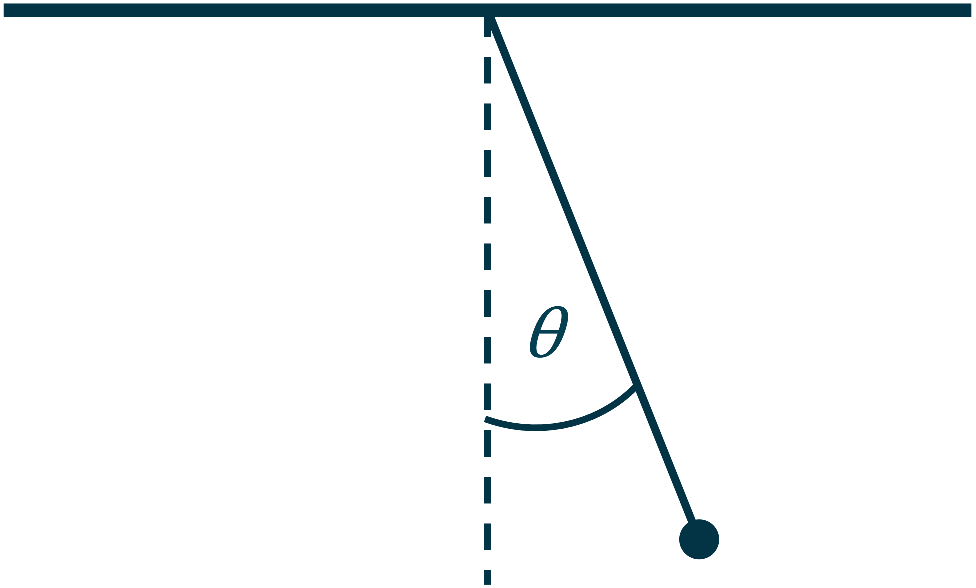
MARS CLIMATE ORBITER SEPTEMBER 23, 1999





A SOFTWARE PROBLEM WITH WRONG METRIC UNITS COST NASA $327 MILLION –NOT A GOOD DAY AT CAPE CANAVERAL

Pendulum equation



$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta$$

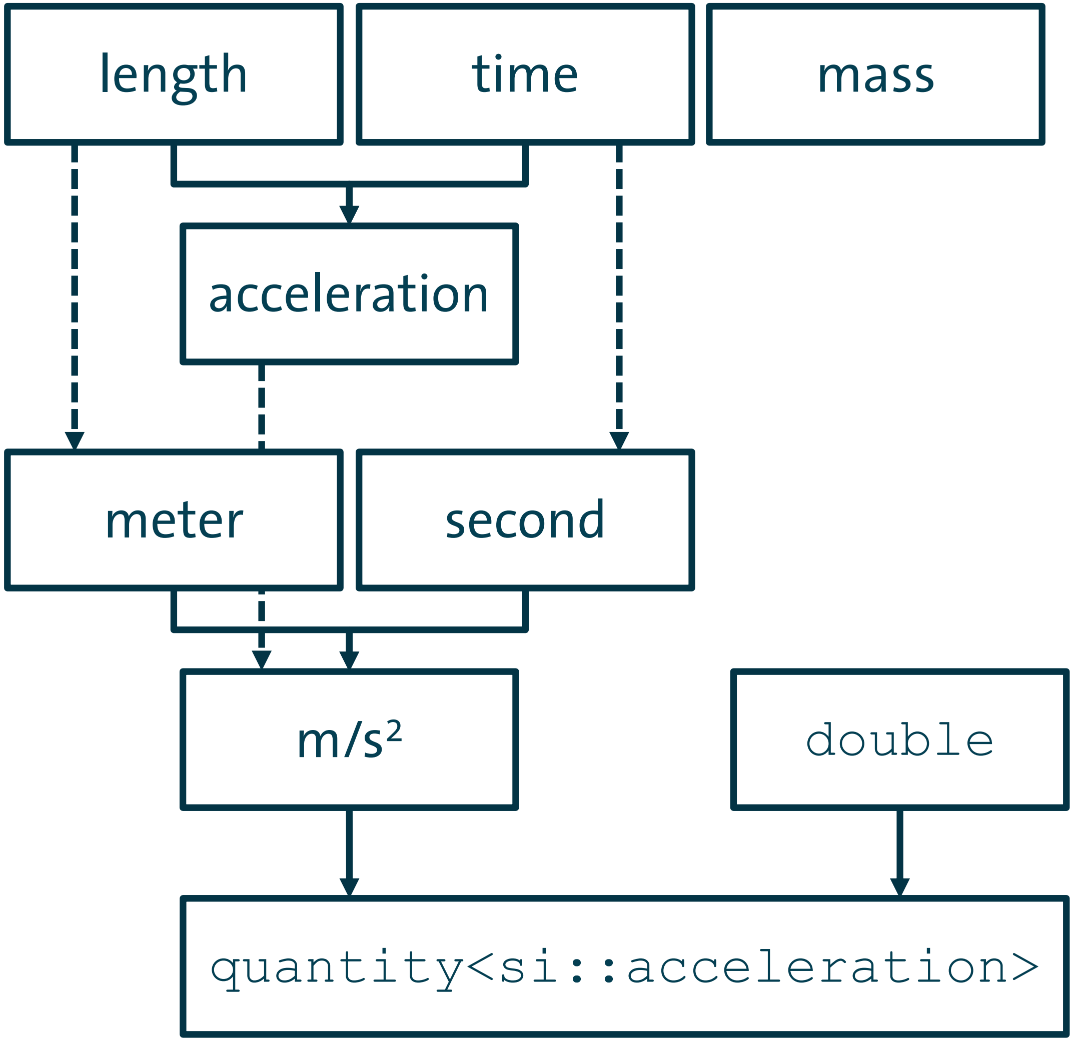
Macho style

```
#include <cmath>
#include <iostream>

int main()
{
  const double g=9.8;   // m/s2
  const double l=50.0;  // cm
  const double dt=0.02; // s

  double theta=30.0;    // degrees
  double theta_v=0.0;   // degrees/s

  std::cout<<"time\tangle\n";
  for(double t=0.0;t<2.0;t+=dt){
    double theta_a=-sin(theta*2.0*3.14159265/360)*g/(l/100.0);
    theta_v+=theta_a*dt;
    theta+=theta_v*dt;
    std::cout<<t<<" s\t"<<theta<<" deg\n";
  }
}
```

Results: go launch the probe!

```
time     angle        time     angle        time     angle        time     angle
0 s      29.9961 deg  0.5 s    28.6336 deg  1 s      24.938 deg   1.5 s    19.1909 deg
0.02 s   29.9882 deg  0.52 s   28.5293 deg  1.02 s   24.7451 deg  1.52 s   18.9242 deg
0.04 s   29.9765 deg  0.54 s   28.4212 deg  1.04 s   24.5488 deg  1.54 s   18.655 deg
0.06 s   29.9608 deg  0.56 s   28.3094 deg  1.06 s   24.3493 deg  1.56 s   18.3833 deg
0.08 s   29.9412 deg  0.58 s   28.1939 deg  1.08 s   24.1466 deg  1.58 s   18.109 deg
0.1 s    29.9177 deg  0.6 s    28.0747 deg  1.1 s    23.9406 deg  1.6 s    17.8324 deg
0.12 s   29.8903 deg  0.62 s   27.9518 deg  1.12 s   23.7315 deg  1.62 s   17.5533 deg
0.14 s   29.859 deg   0.64 s   27.8252 deg  1.14 s   23.5192 deg  1.64 s   17.2719 deg
0.16 s   29.8238 deg  0.66 s   27.695 deg   1.16 s   23.3038 deg  1.66 s   16.9882 deg
0.18 s   29.7846 deg  0.68 s   27.5611 deg  1.18 s   23.0853 deg  1.68 s   16.7021 deg
0.2 s    29.7416 deg  0.7 s    27.4236 deg  1.2 s    22.8637 deg  1.7 s    16.4139 deg
0.22 s   29.6947 deg  0.72 s   27.2824 deg  1.22 s   22.639 deg   1.72 s   16.1234 deg
0.24 s   29.6439 deg  0.74 s   27.1377 deg  1.24 s   22.4114 deg  1.74 s   15.8307 deg
0.26 s   29.5892 deg  0.76 s   26.9894 deg  1.26 s   22.1807 deg  1.76 s   15.5359 deg
0.28 s   29.5307 deg  0.78 s   26.8376 deg  1.28 s   21.9471 deg  1.78 s   15.2389 deg
0.3 s    29.4683 deg  0.8 s    26.6822 deg  1.3 s    21.7106 deg  1.8 s    14.94 deg
0.32 s   29.402 deg   0.82 s   26.5233 deg  1.32 s   21.4712 deg  1.82 s   14.639 deg
0.34 s   29.3319 deg  0.84 s   26.3609 deg  1.34 s   21.2289 deg  1.84 s   14.336 deg
0.36 s   29.258 deg   0.86 s   26.195 deg   1.36 s   20.9837 deg  1.86 s   14.031 deg
0.38 s   29.1802 deg  0.88 s   26.0256 deg  1.38 s   20.7358 deg  1.88 s   13.7242 deg
0.4 s    29.0986 deg  0.9 s    25.8528 deg  1.4 s    20.485 deg   1.9 s    13.4155 deg
0.42 s   29.0132 deg  0.92 s   25.6766 deg  1.42 s   20.2316 deg  1.92 s   13.105 deg
0.44 s   28.924 deg   0.94 s   25.497 deg   1.44 s   19.9754 deg  1.94 s   12.7927 deg
0.46 s   28.8309 deg  0.96 s   25.314 deg   1.46 s   19.7165 deg  1.96 s   12.4787 deg
0.48 s   28.7342 deg  0.98 s   25.1277 deg  1.48 s   19.455 deg   1.98 s   12.163 deg
```

Boost.Units


length
time
mass
acceleration
meter
second
m/s²
double
quantity<si::acceleration>

Boost.Units to the rescue

```
int main()
{
  using boost::units::quantity;
  using boost::units::degree::degree;
  using boost::units::cgs::centimeter;
  using namespace boost::units::si;

  const auto                 g=9.8*meter_per_second_squared;
  const quantity<length>     l{50.0*centimeter};
  const auto                 dt=0.02*second;

  quantity<plane_angle>      theta{30.0*degree};
  quantity<angular_velocity> theta_v;

  std::cout<<"time\tangle\n";
  for(auto t=0.0*second;t<2.0*second;t+=dt){
    auto theta_a=-sin(theta)*radian*g/l;
    theta_v+=theta_a*dt;
    theta+=theta_v*dt;
    std::cout<<t<<"\t"<<
      quantity<boost::units::degree::plane_angle>{theta}<<"\n";
  }
}
```

Results

```
time     angle        time     angle        time     angle        time     angle
0 s      29.7754 deg  0.5 s    -20.226 deg  1 s      -6.8586 deg  1.5 s    27.948 deg
0.02 s   29.3277 deg  0.52 s   -22.0792 deg 1.02 s   -4.27676 deg 1.52 s   26.8919 deg
0.04 s   28.66 deg    0.54 s   -23.7635 deg 1.04 s   -1.66143 deg 1.54 s   25.6326 deg
0.06 s   27.7769 deg  0.56 s   -25.2668 deg 1.06 s   0.966924 deg 1.56 s   24.179 deg
0.08 s   26.6844 deg  0.58 s   -26.5784 deg 1.08 s   3.5877 deg   1.58 s   22.5415 deg
0.1 s    25.3902 deg  0.6 s    -27.6889 deg 1.1 s    6.18037 deg  1.6 s    20.7317 deg
0.12 s   23.9034 deg  0.62 s   -28.5908 deg 1.12 s   8.72467 deg  1.62 s   18.7629 deg
0.14 s   22.2346 deg  0.64 s   -29.2777 deg 1.14 s   11.2008 deg  1.64 s   16.6496 deg
0.16 s   20.3958 deg  0.66 s   -29.7449 deg 1.16 s   13.5898 deg  1.66 s   14.4077 deg
0.18 s   18.4005 deg  0.68 s   -29.9892 deg 1.18 s   15.8731 deg  1.68 s   12.0539 deg
0.2 s    16.2633 deg  0.7 s    -30.009 deg  1.2 s    18.0336 deg  1.7 s    9.60637 deg
0.22 s   14.0004 deg  0.72 s   -29.8042 deg 1.22 s   20.0551 deg  1.72 s   7.08386 deg
0.24 s   11.6288 deg  0.74 s   -29.3761 deg 1.24 s   21.9225 deg  1.74 s   4.50595 deg
0.26 s   9.16663 deg  0.76 s   -28.7276 deg 1.26 s   23.6222 deg  1.76 s   1.89275 deg
0.28 s   6.63292 deg  0.78 s   -27.8632 deg 1.28 s   25.1419 deg  1.78 s   -0.735286 deg
0.3 s    4.04732 deg  0.8 s    -26.7889 deg 1.3 s    26.4707 deg  1.8 s    -3.35756 deg
0.32 s   1.43001 deg  0.82 s   -25.5122 deg 1.32 s   27.5993 deg  1.82 s   -5.95352 deg
0.34 s   -1.1985 deg  0.84 s   -24.0419 deg 1.34 s   28.5199 deg  1.84 s   -8.50289 deg
0.36 s   -3.81762 deg 0.86 s   -22.3887 deg 1.36 s   29.2259 deg  1.86 s   -10.9858 deg
0.38 s   -6.40684 deg 0.88 s   -20.5644 deg 1.38 s   29.7126 deg  1.88 s   -13.3832 deg
0.4 s    -8.94592 deg 0.9 s    -18.5822 deg 1.4 s    29.9767 deg  1.9 s    -15.6766 deg
0.42 s   -11.4152 deg 0.92 s   -16.457 deg  1.42 s   30.0163 deg  1.92 s   -17.8486 deg
0.44 s   -13.7955 deg 0.94 s   -14.2045 deg 1.44 s   29.8312 deg  1.94 s   -19.8829 deg
0.46 s   -16.0687 deg 0.96 s   -11.8417 deg 1.46 s   29.4227 deg  1.96 s   -21.7644 deg
0.48 s   -18.2176 deg 0.98 s   -9.38679 deg 1.48 s   28.7935 deg  1.98 s   -23.4794 deg
```

Extra ball: user-defined literals

User-defined literals

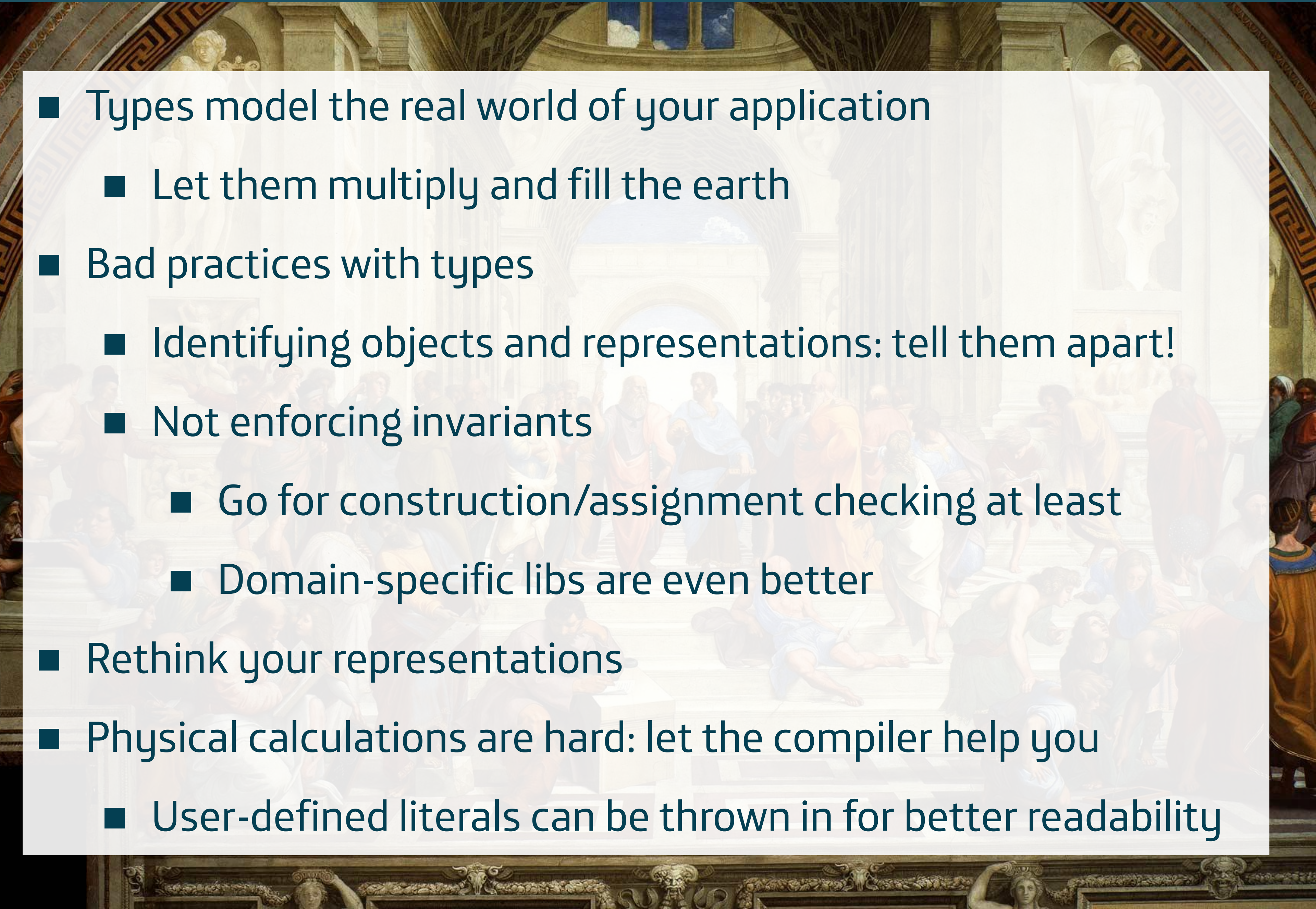
```
quantity<boost::units::si::acceleration> operator"" _m_s2(long double x)
{
  return double(x)*meter_per_second_squared;
}

quantity<boost::units::si::length> operator"" _cm(long double x)
{
  return quantity<boost::units::si::length>{double(x)*centimeter};
}

quantity<boost::units::si::time> operator"" _s(long double x)
{
  return double(x)*second;
}

quantity<boost::units::si::plane_angle> operator"" _deg(long double x)
{
  return quantity<boost::units::si::plane_angle>{double(x)*degree};
}

quantity<boost::units::si::angular_velocity> operator"" _deg_s(long double x)
{
  return quantity<boost::units::si::angular_velocity>{double(x)*degree/second};
}
```

User-defined literals

```
int main()
{
  const auto g=9.8_m_s2;
  const auto l=50.0_cm;
  const auto dt=0.02_s;

  auto       theta=30.0_deg;
  auto       theta_v=0.0_deg_s;

  std::cout<<"time\tangle\n";
  for(auto t=0.0_s;t<2.0_s;t+=dt){
    auto theta_a=-sin(theta)*radian*g/l;
    theta_v+=theta_a*dt;
    theta+=theta_v*dt;
    std::cout<<t<<"\t"<<
      quantity<boost::units::degree::plane_angle>{theta}<<"\n";
  }
}
```

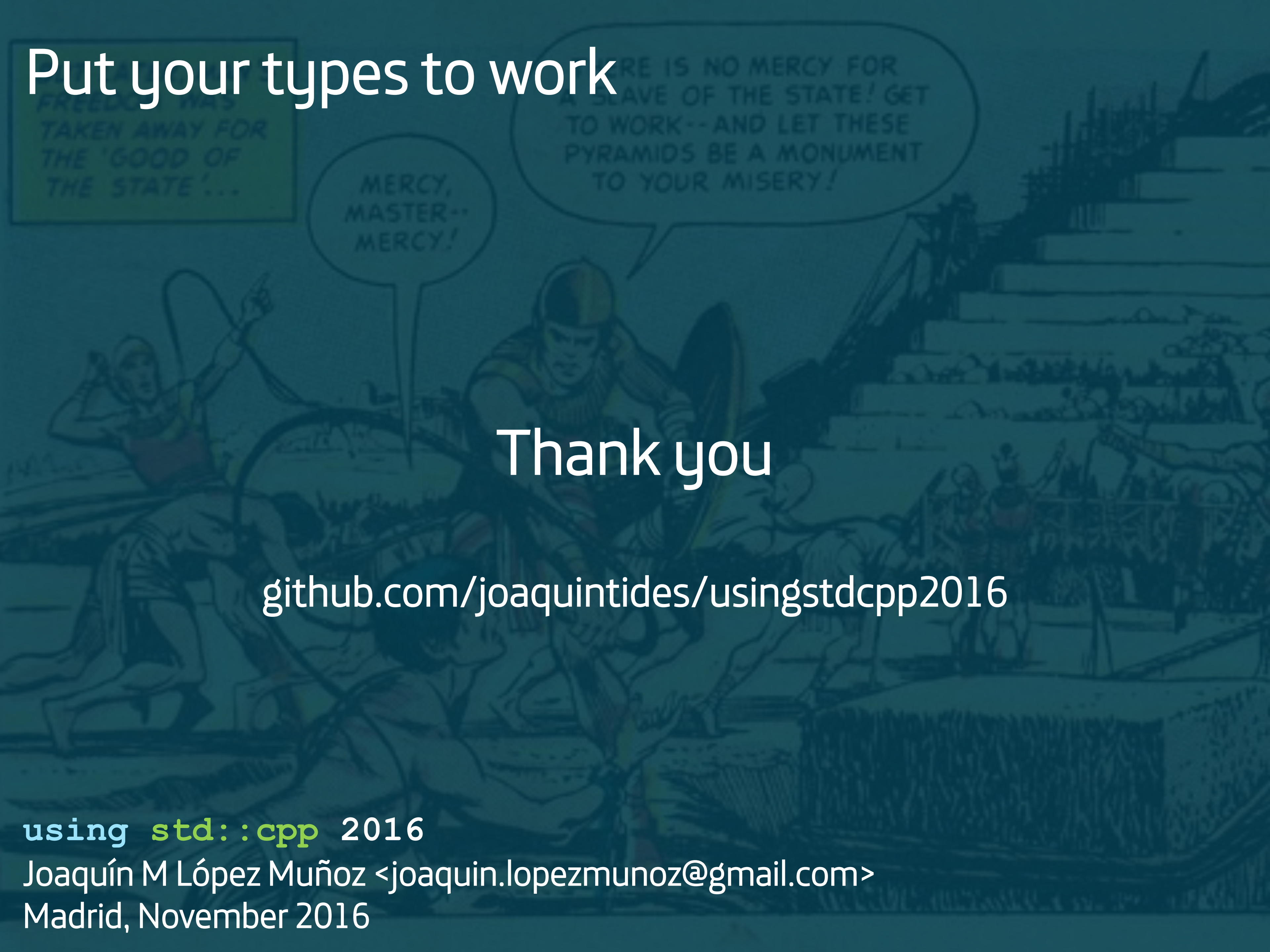
Summary

Summary



- Types model the real world of your application
  - Let them multiply and fill the earth
- Bad practices with types
  - Identifying objects and representations: tell them apart!
  - Not enforcing invariants
    - Go for construction/assignment checking at least
    - Domain-specific libs are even better
- Rethink your representations
- Physical calculations are hard: let the compiler help you
  - User-defined literals can be thrown in for better readability

Put your types to work


FREED WAS TAKEN AWAY FOR THE 'GOOD OF THE STATE'...
MERCY, MASTER-- MERCY!
RE IS NO MERCY FOR A SLAVE OF THE STATE! GET TO WORK--AND LET THESE PYRAMIDS BE A MONUMENT TO YOUR MISERY!

Thank you

github.com/joaquintides/usingstdcpp2016

`using std::cpp 2016`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>
Madrid, November 2016