PRIMEDIC™

Saves Life. Everywhere.

Embedded programming with C++11

Design aims of C++11

Bjarne Stroustrup:

“ *... make C++ even better for embedded system programming ...* ”

Overview

- Key concerns
  - Safety critical
  - Limited resources
  - Long lifetime
  - Many cores
- Last thoughts
  - Myths
  - Facts

Preventing narrowing with **{}-initialization**

3.14159 → 3.~~14159~~

```
double dou= 3.14159;
int a= dou;         // ok
int b(dou);         // ok

int c= {dou};       // error
int d{dou};         // error

int8_t f= {2011};   // error
int8_t g= {14};     // ok
```

→ narrowing conversion from double to int

Assertions with **type traits and static_assert**

- type traits performs at compile time
  - type information
  - type comparison
  - type modification
- static_assert
  - validate expressions at compile time

Assertions with type **traits and static_assert**

```
template <typename S, typename D>
void smallerAs(S s,D d){
  static_assert(sizeof(S) <= sizeof(D),"S is too big");
}

smallerAs(1.0,1.0L);
smallerAs(1.0L,1.0);      // with S= long double; D= double
                          // S is too big

template <typename T>
T fac(T a){
  static_assert(std::is_integral<T>::value,"T not integral");
}

fac(10);
fac(10.1);                // with T= double; T not integral
```

Respect the unit with **user-defined literals**

- Syntax: <built_in literal>+_ + <suffix>
  - integer literals: 101010_b
  - floating point literals: 123.45_km
  - string literals: “hello”_i18n
  - character literals: ’a’_NoIdea

Respect the unit with **user-defined literals**

```
using namespace Unit;
using namespace std;

int main(){

  cout << 1.0_km + 2.0_dm +  3.0_dm + 4.0_cm; // 1000054 cm

  MyDist myDist= 10345.5_dm + 123.45_km - 1200.0_m + 150000.0_cm;

  cout << myDist;                             // 1.24785e+07 cm
}
```

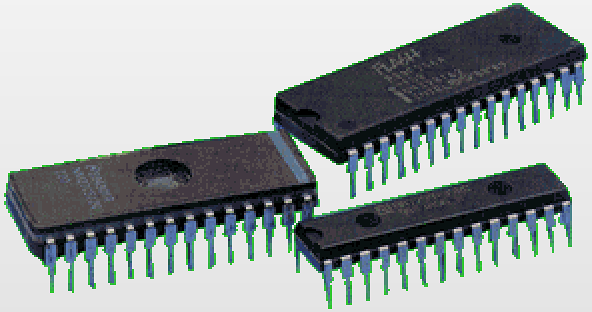
Respect the unit with **user-defined literals**

```
namespace Unit{
  MyDist operator "" _km(long double k){
    return MyDist(100000*k);
  }
  MyDist operator "" _m(long double m){
    return MyDist(100*m);
  }
  MyDist operator "" _dm(long double d){
    return MyDist(10*d);
  }
  MyDist operator "" _cm(long double c){
    return MyDist(c);
  }
}
```

Respect the unit with **User-defined literals**

```
class MyDist{
  private:
    long double cm;
  public:
    MyDist(long double i):cm(i){}
    friend MyDist operator +(const MyDist& a, const MyDist& b){
      return MyDist(a.cm + b.cm);
    }
    friend MyDist operator -(const MyDist& a,const MyDist& b){
      return MyDist(a.cm - b.cm);
    }
    friend std::ostream& operator<< (std::ostream &out, const MyDist&
    myDist){
      out << myDist.cm << " cm";
      return out;
    }
};
```

Evaluate at compile time with **constexpr**



- can be evaluated at compile time

  ➡ stored in ROM
- three forms
  - variables
  - functions
  - user-defined types

Evaluate at compile time with **constexpr**

```
constexpr int myConstExpr= 2;
constexpr int square(int i){ return i*i; }

struct MyInt{
  int myInt;
  constexpr MyInt(int i):myInt(i){}
  constexpr int multiplyBy(int i){ return i*myInt; }
};

constexpr MyInt myInt(5);
constexpr int res= myInt.multiplyBy(myConstExpr);

static_assert(myInt.multiplyBy(2) == 10,"error");
static_assert(res == 10,"error");
std::cout << myInt.multiplyBy(square(5)) << std::endl;
```

Be fast with **generalized POD‘s**

- has to be trivial
- has standard layout
- members and base classes must also be POD’s
  - fast manipulation like a C struct
    - arrays of POD’s can be copied by block
  - static initialization

Be fast with **generalized POD‘s**

```
struct Base{};

struct Pod: Base {
  int a;
  Pod()= default;
  int getA() const { return a;}
};

. . .

std::cout << std::is_pod<int>::value;          // true
std::cout << std::is_pod<std::string>::value;  // false
std::cout << std::is_pod<Pod>::value;          // true
```

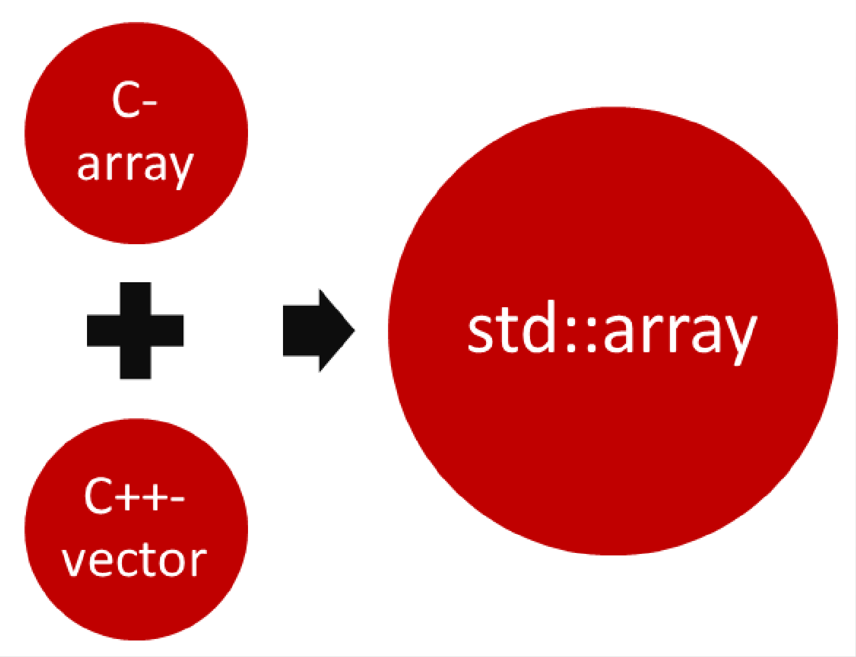
Be fast with **generalized POD‘s**

```
struct NotTrivial{
  NotTrivial(){}
};
class NotStandLay{
  int a;
public:
  int b;
};

std::cout << std::is_pod<NotTrivial>::value           // false
std::cout << std::is_trivial<NotTrivial>::value;      // false
std::cout << std::is_standard_layout<NotTrivial>::value; // true

std::cout << std::is_pod<NotStandLay>::value;         // false
std::cout << std::is_trivial<NotStandLay>::value;     // true
std::cout << std::is_standard_layout<NotStandLay>::value;//false
```

Slim and fast with **std::array**



- homogeneous container of fixed length
- combines the performance of a C-Array with the interface of a C++-Vector
- no heap allocation

Slim and fast with **std::array**

```
std::array<int,6> arrCpp{{1,2,3,4,5,6}};
int arrC[]= {1,2,3,4,5,6};

static_assert(sizeof(arrCpp) == 6*sizeof(int),"wrong size");
static_assert(sizeof(arrCpp) == sizeof(arrC),"size differs");

             // 21
std::cout << std::accumulate(arrCpp.begin(),arrCpp.end(),0);

             // 720
std::cout << std::accumulate(arrCpp.begin(),arrCpp.end(),1,
              [](int a,int b){return a*b;});
```

Cheap moving with **move semantic**

- cheap moving instead of expensive copying
  - performance
  - no memory allocation and deallocation

  → predictability
- implementing save “move-only” types
  - unique_ptr, files, locks and tasks

Cheap moving with **move semantic**

```
std::vector<int> a, b;
swap(a,b);

template <typename T>
void swap(T& a, T& b){
  T tmp(a);
  a= b;
  b= tmp;
}

template <typename T>
void swap(T& a, T& b){
  T tmp(std::move(a));
  a= std::move(b);
  b= std::move(tmp);
}
```

- `T tmp(a);`
  - allocate tmp and each element of tmp
  - copy each element of a to tmp
  - deallocate tmp and each element of tmp

- `T tmp(std::move(a));`
  - adjust a pointer of tmp to the data of a

Preserve the nature with **perfect forwarding**

- pass the arguments while preserving the lvlue/rvalue nature of the arguments
- use case
  - factory functions
  - constructors

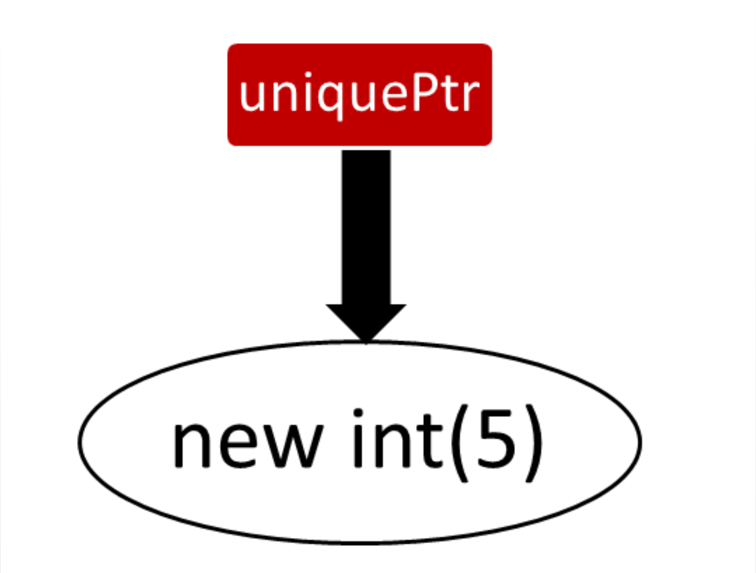
→ chaining of move semantic
forwarding of move-only types

Preserve the nature with **perfect forwarding**

```
template <typename T, typename T1>
T createT(T1&& t1){
  return T(std::forward<T1>(t1));
}
int lValue= createT<int>(2011);
int i= createT<int>(lValue);

struct NeedOnlyMove{
  NeedOnlyMove(OnlyMove){};
};
struct OnlyMove{
  OnlyMove()= default;
  OnlyMove(const OnlyMove&)= delete;
  OnlyMove& operator= (const OnlyMove&)= delete;
  OnlyMove(OnlyMove&&)= default;
  OnlyMove& operator= (OnlyMove&&)= default;
};
NeedOnlyMove nOnlyMove2=  createT<NeedOnlyMove>(OnlyMove());
```

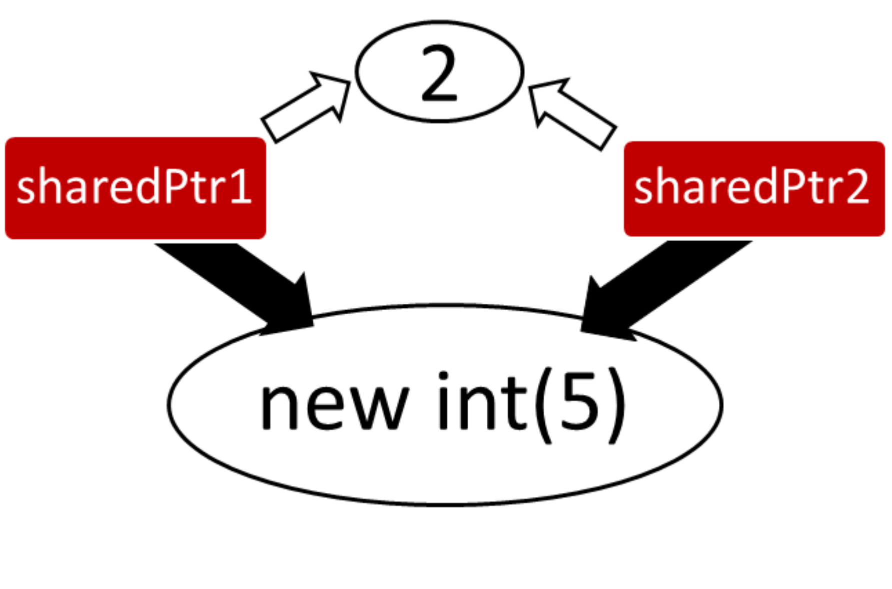
Explicit ownership with **std::unique_ptr**



- explicit ownership
- only moveable
- support arrays

- create and forget
- minimal space and time overhead
- support special allocation strategies

Shared ownership with **std::shared_ptr**



- shared ownership
- has a reference counter and a handle to his resource
- manage the reference counter and the resource

→

- managing overhead in time and space
- saves memory
- have to deal with cycles

performance matters

```
auto st = std::chrono::system_clock::now();

for (long long i=0 ; i < 100000000; ++i){
  int* tmp(new int(i));
  delete tmp;
  // std::unique_ptr<int> tmp(new int(i));
  // std::shared_ptr<int> tmp(new int(i));
  // std::shared_ptr<int> tmp= std::make_shared<int>(i);
}

std::chrono::duration<double> dur=std::chrono::system_clock::now() - st();
std::cout << dur.count();
```

pointer type	real hardware	virtualization
native	3.0 sec.	5.7 sec.
std::unique_ptr	2.9 sec.	5.7 sec.
std::shared_ptr	6.0 sec.	11.8 sec.
std::make_shared		6.5 sec.

Still missing …

- Multiple cores
  - memory model, atomics, thread management
- Limited resource time
  - std::tuple and std::forward_list
  - unordered containers
- Limited resource memory
  - alignment support
- Safety critical
  - scoped enums and nullptr
  - auto_ptr deprecated
- . . .

Myths about C++

Myths about C++

- Templates causes code bloat.
- Objects have to be created on the heap.
- Exceptions are expensive.
- C++ is too slow and needs too much memory.
- C++ is too dangerous for safety critical systems.
- In C++ you have to program object oriented.
- You can use C++ only for applications.
- The iostream library is too big, the STL library too slow.

→ C++ is a nice toy. Be we are dealing with the serious problems.

Facts about C++

Facts about C++

- MISRA C++
  - Motor Industry Software Reliability Association
  - guidelines for C++ in critical(embedded) systems
- TR18015.pdf
  - Technical report on C++ performance
  - special focus on embedded systems
  - refute the myths

→ both based on C++03, but C++11 is still better for the embedded programming

PRIMEDIC™
Saves Life. Everywhere.

Thank you for your attention

Rainer Grimm
Metrax GmbH
www.primedic.com

Phone +49 (0)741 257-0
Rainer.Grimm@primedic.com