C++1* Tech Talks

Initialization, Construction and Deconstruction

Hannes Hauswedell

and Wikipedia, and Stackoverflow...

REINERT LAB
GENOMICS, PROTEOMICS, ALGORITHMS

Max Planck Institute for Molecular Genetics

Freie Universität Berlin

September 16, 2016

Initialization

Default initialization	`std::string s;`	
Value initialization	`std::string s{};`	`std::string s();`
Direct initialization	`std::string s{"Foo"};`	`std::string s("Foo");`
Copy initialization	`std::string s = "Foo";`	`std::string s(s2);`
Direct List initialization	`std::string s{'F','o','o'};`	
Copy List initialization	`std::string s = {'F','o','o'};`	
Aggregate initialization	`char a[3] = {'a', 'b'};`	`T t{'F',4};`
Reference initialization	`char& c = a[0];`	

Default Initialization – no () or {}

```
std::string s;
std::string * p_s = new std::string;
// -> default constructed

std::string sa[5];
std::string p_sa = new std::string[5];
// -> elements are default initialized which means default constructed

int i;
int *p_i = new int;
// -> not initialized at all

int ia[5];
int p_ia = new int[5];
// -> elements are default initialized which means not initialized
```

Summary: default constructor or no initialization

Value Initialization – empty () or {}

```
std::string s();
std::string s{};
std::string * p_s = new std::string();
std::string * p_s = new std::string{};
// -> std::string has user-provided default constr.
// => default initialized which means default constructed

int i();
int i{};
int *p_i = new int();
int *p_i = new int{};
// POD, so zero-initialized, which means == 0

T t{}; // type with implicit default constructor
T t();
//...
// first zero-initialized, then default initialized (dunno why)
```

Summary: zero initializes PODs and POD members of aggregate types; default initializes the rest

Direct Initialization – () or {} with arguments or casts

```
std::string s("Foo");
std::string s{"Foo"};
std::string * p_s = new std::string("Foo");
std::string * p_s = new std::string{"Foo"};
// -> best matching constructor is selected for initialization

int i(0);
int i{7.3};
int *p_i = new int(-5.3);
int *p_i = new int{7};
// non-class types try conversions from the argument to the type
```

Summary:

- “regular” constructor selection
- does not apply to arrays
- narrowing conversions allowed

Copy Initialization – using the assignment operator

```
1 std::string s = "Foo";
  // -> best matching constructor is selected for initialization
3
  int i = 0;
5 int i = 7.3;
  // non-class types try conversions from the argument to the type
```

Summary:

- only non-explicit constructors used
- otherwise same as direct initialization, but with $=$
- avoid this except for PODs, use Direct Initialization instead

Aggregate initialization

```
struct S {
    int x;
    struct Foo {
        int i;
        int j;
        int a[3];
    } b;
};

S s1 = {1, {2, 3, {4, 5, 6} } };      // copy-initializes elements from args
S s2 = {1,  2, 3,  4, 5, 6    };      // with brace elision
S s3   {1, {2, 3, {4, 5, 6} } };      // using direct-list-initialization syntax
S s4   {1,  2, 3,  4, 5, 6    };      // error in C++11, but okay since C++14

S s5   {1, {2, 3, {4, 5   } } };      // last value in b.a will be value/zero-initialized
S s6   {1, {2, 3, {4, 5, 6, 7 } } }; // compile-time error, ill-formed

int ai[] = {1, 2.0 };           // narrowing conversion: okay in C++03, error since C++11
```

Summary:

- makes many constructors redundant, simplifies code
- prefer the syntax without assignment operator and avoid brace elision

List initialization / uniform initialization syntax – everything with {}

```
T object{arg1, arg2, ...};          // direct list initialization
T object = new T{arg1, arg2, ...};  // direct list initialization
T object = {arg1, arg2, ...};       // copy list initialization
```

Summary:

- generalized uniform syntax by using braces with 0-n args `{ arg1, arg3, ... }`
- zero `args` and non-aggregate type → value initialization
- one `arg` and has same type as `T` → direct / copy initialization
- `T` is an aggregate Type → aggregate initialization
- one `arg` and different type or more args **and** `T` is non-aggregate class type:
  - look for constructors of `T` with `std::initializer_list` as parameter
  - then look for constructors with set of arguments

Uniform initialization

```
1  struct AggregateType
   {
3      float x_;
       int y_;
5  };
   AggregateType scalar{0.43f, 10};
7



9
   struct NonAggregateType
11 {
       int x_;
13     double y_;
       NonAggregateType() {}
15     NonAggregateType(int x, double y) : x_{x}, y_{y} {}
   };
17 NonAggregateType var2{2, 4.3};

19 //.
```

Some training

```
struct AggregateType
{
    float x_ = 0.1f;
    int y_;
};
AggregateType a1;                       // a1.x_ ==    ?, a1.y_ == ?



struct NonAggregateType
{
    int x_ = 7;
    double y_;
    NonAggregateType() {}
    NonAggregateType(int x, double y) : x_{x}, y_{y} {}
};
NonAggregateType n1;                    // n1.x_ == ?, n1.y_ == ?

//.
```

Some training

```
struct AggregateType
{
    float x_ = 0.1f;
    int y_;
};
AggregateType a1;                   // a1.x_ ==    ?, a1.y_ == ?
AggregateType a2{};                 // a1.x_ ==    ?, a1.y_ == ?
AggregateType a3{0.43f, 10};        // a1.x_ ==    ?, a1.y_ == ?

struct NonAggregateType
{
    int x_ = 7;
    double y_;
    NonAggregateType() {}
    NonAggregateType(int x, double y) : x_{x}, y_{y} {}
};
NonAggregateType n1;                // n1.x_ == ?, n1.y_ == ?
NonAggregateType n2{};              // n1.x_ == ?, n1.y_ == ?
NonAggregateType n3{2, 4.3};        // n1.x_ == ?, n1.y_ == ?
```

Some training

```
struct AggregateType
{
    float x_ = 0.1f;
    int y_;
};
AggregateType a1;                   // a1.x_ == 0.1 , a1.y_ == unitialized
AggregateType a2{};                 // a1.x_ == 0.1 , a1.y_ == 0
AggregateType a3{0.43f, 10};        // a1.x_ == 0.43, a1.y_ == 10

struct NonAggregateType
{
    int x_ = 7;
    double y_;
    NonAggregateType() {}
    NonAggregateType(int x, double y) : x_{x}, y_{y} {}
};
NonAggregateType n1;                // n1.x_ == 7, n1.y_ == unitialized
NonAggregateType n2{};              // n1.x_ == 7, n1.y_ == unitialized
NonAggregateType n3{2, 4.3};        // n1.x_ == 2, n1.y_ == 4.3
```

Summary

- design classes without constructors and use aggregate initialization (whenever possible)
- for default values, use member initializers
- always prefer brace initializers and don't use parantheses anymore
- be careful with the empty brace initializer (PODs/aggregates vs non-aggregates)

Bonus: Are () and {} initialization always the same for one type?

```
1 std::vector<std::string> v{100};
  std::vector<std::string> v(100);
3
  std::vector<int> v{100};
5 std::vector<int> v(100);
```

Summary

- design classes without constructors and use aggregate initialization (whenever possible)
- for default values, use member initializers
- always prefer brace initializers and don't use parantheses anymore
- be careful with the empty brace initializer (PODs/aggregates vs non-aggregates)

Bonus: Are () and {} initialization always the same for one type?

```
1 std::vector<std::string> v{100};   // v.size() == 100, v[0] == ""
  std::vector<std::string> v(100);   // v.size() == 100, v[0] == ""
3 // but...
  std::vector<int> v{100};           // v.size() == 1,   v[0] == 100
5 std::vector<int> v(100);           // v.size() == 100, v[0] == 0
```