21-initialization

June 2, 2024

1 Uniform initialization?

The history of the language has led to multiple ways of initializing different types of variables and objects. This confuses the programmer, harms the readability of the code, and **implies problems in the use of templates**.

1.1 C++ 98: generalization of initialization by parentheses

[1]:
```
class MyInt
 {
  public :
    MyInt( int i ) : m_data(i) {}
  private :
    int m_data ;
 } ;
```

[2]:
```
MyInt i2(42)  ;
```

[3]:
```
int i1(42)    ;
```

But this does not solve some parsing problems, such as the infamous **most vexing parse**.

1.2 C++11: generalization of initialization by curly braces

It was already possible for aggregate types

[4]:
```
struct St {
  int x ;
  double z ;
} ;
```

[5]:
```
St s1 = { 4, 5.3 } ;
```

[6]:
```
int t[] = { 3, 5, 9 } ;
```

C++11 enlarges this for all types, with or without =

[8]:
```
int i1 = { 42 } ;
int i2 { 42 }   ;
MyInt i3 = { 42 }  ;
```

```
MyInt i4 { 42 }  ;
```

Improvment: using empty braces forces initialization to the "null" value of the type

[9]:
```
#include <iostream>
int n {} ;
std::cout << n << std::endl ;
```

```
0
```

Improvement: narrowing of values results in compiler warning or error

[1]:
```
int x, y, z ;
//...
float sum1 {x+y+z} ;
```

```
input_line_7:4:13: error: non-constant-expression

cannot be narrowed from type 'int' to 'float' in initializer list

[-Wc++11-narrowing]
float sum1 {x+y+z} ;
            ^~~~~

input_line_7:4:13: note: insert an explicit cast to
silence this issue
float sum1 {x+y+z} ;
            ^~~~~

            static_cast<float>( )
```

```
Interpreter Error:
```

1.3 The new collection `std::initializer_list<T>`

Default type of values in curly braces

[1]:
```
#include <iostream>
```

[2]:
```
auto params = { 1, 2, 3, 4, 5 } ;
for ( auto param : params )
 { std::cout << param << " " ; }
std::cout << std::endl ;
```

```
1 2 3 4 5
```

You can give your class a constructor from `std::initializer_list<T>`

[3]:
```
#include <iostream>
#include <vector>
```

```
template <typename T>
class MyCollection {
  public:
    MyCollection( std::initializer_list<T> const & elements )
     : m_elements(elements) {}
    void display()
     {
      for ( auto element : m_elements )
       { std::cout<<element<<" " ; }
      std::cout<<std::endl ;
     }
  private :
    std::vector<T> m_elements ;
} ;
```

[4]:
```
MyCollection col { 1, 2, 3, 4, 5 } ;
col.display() ;
```

```
1 2 3 4 5
```

Beware if you mix it with other constructors : std::initializer_list<T> has priority.

[8]:
```
#include <iostream>

template <typename T>
class MyCollection {
  public:
    MyCollection( int size, T value )
     { std::cout<<"MyCollection( int size, int value )"<<std::endl ; }
    MyCollection( std::initializer_list<T> const & )
     { std::cout<<"MyCollection( std::initializer_list<T> const & )"<<std::endl␣
  ↪; }
} ;
```

[9]:
```
MyCollection col { 1, 2, 3 } ;
```

```
MyCollection( std::initializer_list<T> const & )
```

[10]:
```
MyCollection col { 1, 2 } ;
```

```
MyCollection( std::initializer_list<T> const & )
```

[11]:
```
MyCollection col ( 1, 2 ) ;
```

```
MyCollection( int size, int value )
```

This is the case for std::vector

[12]:
```
#include <iostream>
#include <vector>
```

[13]:
```
std::vector<int> col { 3, 1 } ;
for ( int elem : col )
{ std::cout<<elem<<" " ; }
```

```
3 1
```

[14]:
```
std::vector<int> col ( 3, 1 ) ;
for ( int elem : col )
{ std::cout<<elem<<" " ; }
```

```
1 1 1
```

2 Take away

- The use of `{}` is today the most usable way of initialization.
- I do not recommend `= {}`, which looks like affectation.
- We still need to initialize with `()` in some situations.
- I use `{}` when the arguments are values to be embedded, `()` in other cases.

3 Questions ?

4 Exercise

In the class below, replace the `add()` method with a constructor from a `std::initializer_list`, and simplify `main`.

[15]:
```
%%file tmp.initialisation.cpp

#include <iostream>
#include <string>
#include <vector>

class Sentence {
  public :
    void add( char const * word )
     { m_words.emplace_back(static_cast<std::string>(word)) ; }
    auto begin() const { return m_words.begin() ; }
    auto end() const { return m_words.end() ; }
  private :
    std::vector<std::string> m_words ;
} ;

std::ostream & operator<<( std::ostream & os, Sentence const & s ) {
  for ( auto const & word : s )
   { os<<word<<" " ; }
  return os ;
}
```

```
int main() {
  Sentence s ;
  s.add("Hello") ;
  s.add("world") ;
  s.add("!") ;
  std::cout<<s<<std::endl ;
}
```

```
Writing tmp.initialisation.cpp
```

[16]:
```
!rm -f tmp.initialisation.exe && g++ -std=c++17 tmp.initialisation.cpp -o tmp.
↪initialisation.exe
```

[17]:
```
!./tmp.initialisation.exe
```

```
Hello world !
```