31-rvalue-references

June 2, 2024

1 Modern C++: rvalue references and data moves

1.1 Reminders on references

A reference is an alias to another variable

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

int var {42} ;
int & ref {var} ;
ref = 99 ;

std::cout << var << std::endl ;
```

```
99
```

A reference cannot bind to a temporary value

[2]:
```
int & i {42} ; // Won't compile
```

```
input_line_9:2:8: error: non-const lvalue reference to
type 'int' cannot bind to an initializer list temporary
 int & i {42} ; // Won't compile
       ^ ~~~~
```

```
Interpreter Error:
```

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

void print( std::string & s )
 { std::cout<<s<<std::endl ; }

print("hello") ; // Won't compile
```

```
input_line_11:4:2: error: no matching function for call
to 'print'
 print("hello") ; // Won't compile
```

```
 ^~~~~
input_line_11:1:6: note: candidate function not viable:
no known conversion from 'const char [6]' to 'std::string &' (aka
'basic_string<char> &') for 1st argument
void print( std::string & s )
     ^
```

```
Interpreter Error:
```

Unless for references to constants ***restart kernel***

[1]:
```
int const & i {42} ; // OK
```

[2]:
```
#include <iostream>
#include <string>

void print_const( std::string const & s )
 { std::cout<<s<<std::endl ; }

print_const("hello world") ; // OK
```

```
hello world
```

1.2 Modern C++: rvalue references

An rvalue reference can only bind to a temporary value

[3]:
```
int && i {42} ; // OK
```

[4]:
```
int j {42} ;
int && k {j} ; // Won't compile
```

```
input_line_11:3:8: error: rvalue reference to type
'int' cannot bind to lvalue of type 'int'
int && k {j} ; // Won't compile
       ^ ~~~
```

```
Interpreter Error:
```

In case of temporary value, a function with rvalue reference will be prefered

[5]:
```
#include <iostream>
```

[6]:
```
void process( int & value )       { std::cout<<"process(&): "<<value<<std::endl␣
 ↪; }
```

[7]:
```
void process( int const & value ) { std::cout<<"process(const &): "<<value<<std:
 ↪:endl ; }
```

[8]:
```
void process( int && value )      { std::cout<<"process(&&): "<<value<<std::
 ↪endl ; }
```

[9]:
```
int i {42} ;
int const & ic {i} ;
```

[10]:
```
process(i) ;
```

```
process(&): 42
```

[11]:
```
process(ic) ;
```

```
process(const &): 42
```

[12]:
```
process(42) ;
```

```
process(&&): 42
```

1.3 An opportunity of optimization

Utility function

[13]:
```
#include <iostream>
#include <string>
```

[14]:
```
template < typename Collection >
void display( std::string const & title, Collection const & col )
 {
  std::cout<<title<<":" ;
  for ( auto elem : col )
   { std::cout<<" "<<elem ; }
  std::cout<<std::endl ;
 }
```

This function sorts and displays a collection of integers

[15]:
```
#include <vector>
```

[16]:
```
void process( std::vector<int> const & col )
 {
  std::vector<int> copy(col);
  std::sort(copy.begin(),copy.end()) ;
  display("process(const &)",copy) ;
 }
```

[17]:
```
std::vector<int> const vic {1,3,2} ;
process(vic) ;
```

```
process(const &): 1 2 3
```

[18]:
```
std::vector<int> vi {6,4,5} ;
process(vi) ;
```

```
process(const &): 4 5 6
```

[19]:
```
process({9,8,7}) ;
```

```
process(const &): 7 8 9
```

This new version avoids copying, but does not accept temporary collection

[20]:
```
void process( std::vector<int> & col )
 {
  std::sort(col.begin(),col.end()) ;
  display("process(&)",col) ;
 }
```

[21]:
```
const std::vector<int> cvi {1,3,2} ;
process(cvi) ;
```

```
process(const &): 1 2 3
```

[22]:
```
std::vector<int> vi {6,4,5} ;
process(vi) ;
```

```
process(&): 4 5 6
```

[23]:
```
process({9,8,7}) ;
```

```
process(const &): 7 8 9
```

To avoid copying the temporary collection, rvalue references must be used

[24]:
```
void process( std::vector<int> && col )
 {
  std::sort(col.begin(),col.end()) ;
  display("process(&&)",col) ;
 }
```

[25]:
```
std::vector<int> const cvi {1,3,2} ;
process(cvi) ;
```

```
process(const &): 1 2 3
```

[26]:
```
std::vector<int> vi {6,4,5} ;
process(vi) ;
```

```
process(&): 4 5 6
```

[27]:
```
process({9,8,7}) ;
```

```
process(&&): 7 8 9
```

1.4 When a function call another function

A rvalue reference is not an rvalue If a variable is an rvalue reference, beginners generally assume that it is itself some kind of rvalue, and will be considered as so when transmitted to another function. **IT IS NOT**.

In the code below, `col` refers to a rvalue (`{9,8,7}`), but `col` itself is not a temporary value, as testified by the call to `process()`, which the compiler interpret as `process( std::vector<int> & )`.

[32]:
```
void super_process( std::vector<int> && col )
 { process(col) ; }
```

[33]:
```
super_process({9,8,7}) ;
```

```
process(&): 7 8 9
```

Transform any value into an rvalue with `std::move` When you want a value, any, to be considered as an rvalue, you can simply call `std::move` on it. This is a way to say that the value is somehow "temporary", "movable", soon destructed, and can be directly reused and modified as needed.

[34]:
```
void super_process( std::vector<int> && col )
 { process(std::move(col)) ; }
```

[35]:
```
super_process({9,8,7}) ;
```

```
process(&&): 7 8 9
```

1.5 Classes and movable objects

When a class manage some dynamic resource, it is useful to add a *move constructor*

```
class A {
  public :
    A() : m_data {new int[1000000]} {}
    A( A const & other ) : m_data {new int[1000000]}
     { std::copy(other.m_data,other.m_data+1000000,m_data) ; }
    A( A && other ) : m_data {other.m_data} { other.m_data = 0 ; }
    ~A() { delete [] m_data; }
  private :
    int * m_data ;
} ;
```

In fact, in case of dynamic resources, there are eventually 5 methods to define ("The Big Five")

```
A( A const & )              ; // copy constructor
A( A && )                   ; // move constructor
A & operator=( A const & ) ; // copy assignment operator
A & operator=( A && )       ; // move assignment operator
~A()                        ; // destructor
```

If you choose to prohibit the copy: the class is said *move-only*

```
public :
  A( A && )                   ;
  A & operator=( A && )       ;
  ~A()                        ;
private :
  A( A const & )              ;
  A & operator=( A const & ) ;
```

2 Questions ?

3 Exercise

The class below mimics a very simplified `std::string`. Add a move constructor, and ensure that it is used by `main()`.

[38]:
```
%%file tmp.references.cpp

#include <cstring>
#include <iostream>

class Text
 {
  public :

    Text( char const * str )
     : m_size {std::strlen(str)+1},
       m_data {new char [m_size]}
     { std::copy(str,str+m_size,m_data) ; }

    Text( Text const & t )
     : m_size {t.m_size},
       m_data {new char [m_size]}
     {
      std::cout<<"copy constructor"<<std::endl ;
      std::copy(t.m_data,t.m_data+m_size,m_data) ;
     }

    ~Text()
     { delete [] m_data ; }
```

```
    unsigned int size()
     { return m_size ; }

    char & operator[]( unsigned int i )
     { return m_data[i] ; }

    friend std::ostream & operator<<( std::ostream & os, Text const & t )
     { return os<<t.m_data ; }

  private :

    std::size_t m_size ;
    char * m_data ;
 } ;

Text hello()
 { return "hello" ; }

Text uppercase( Text t )
 {
  for ( unsigned int i=0 ; i<t.size() ; ++i )
   { t[i] = std::toupper(t[i]) ; }
  return t ;
 }

int main()
 {
  std::cout<<uppercase(hello())<<std::endl ;
  return 0 ;
 }
```

```
Overwriting tmp.references.cpp
```

[39]:
```
!rm -f tmp.references.exe && g++ -std=c++17 tmp.references.cpp -o tmp.
↪references.exe
```

[40]:
```
!./tmp.references.exe
```

```
copy constructor
HELLO
```