* To understand move semantics and perfect forwarding, let’s create a class DoubleVec which holds array of doubles.
* This class is having...
  1. Copy constructor
  2. Move constructor

|  |
| --- |
| class DoubleVec {  int size;  double\* arr; // A very big array of doubles.  public:  DoubleVec(int size): size(size) {  arr = new double[size];  for (int i = 0; i < size; i++) { arr[i] = i; }  }  DoubleVec(const DoubleVec& rhs) { // Copy constructor  *cout* << "Copy Constructor\n";  size = rhs.size;  arr = new double[size];  for (int i = 0; i < size; i++) { arr[i] = rhs.arr[i]; }  }  DoubleVec(DoubleVec&& rhs) { // Move constructor  *cout* << "Move Constructor\n";  size = rhs.size;  arr = new double[size];  for (int i = 0; i < size; i++) { arr[i] = rhs.arr[i]; }  rhs.size = 0;  rhs.arr = nullptr;  }  void print() {  for (int i = 0; i < size; i++)  *cout* << arr[i] << " ";  }  ~DoubleVec() { delete arr; }  }; |

* Let’s add a function which creates this DoubleVec object and returns it.

|  |
| --- |
| DoubleVec createDoubleVec(int size = 50) {  *cout* << "In createDoubleVec\n";  return DoubleVec(size);  } |

* Let’s add another function which prints this DoubleVec object using the print method.

|  |
| --- |
| void foo(DoubleVec v) {  *cout* << "Printing the vector: ";  v.print();  *cout* << "\n";  } |

* Let’s have another function called jumpFunction() which is a templated function and takes an argument and calls the foo function.

|  |
| --- |
| template<typename T>  void jumpFunction(T arg) {  *cout* << "In jumpFunction\n";  foo(arg);  } |

* If we see this jumpFunction(), it is just forwarding the arguments which it has received to foo() function.
* How the jumpFunction() should pass over its arguments to foo() function?
  1. No costly and unnecessary copy construction of DoubleVec should be made.
  2. rvalue should be forwarded as rvalue and lvalue should be forwarded as lvalue.
* Let’s see what happens when we call this jumpFunction() using both lvalue and rvalues.

|  |
| --- |
| int main(void) {  DoubleVec vec = createDoubleVec(20);  *cout* << "Calling jumpFunction with vec (lvalue).\n";  jumpFunction(vec);  *cout* << "Calling jumpFunction with createDoubleVec (rvalue).\n";  jumpFunction(createDoubleVec(20));  return 0;  } |

* Let’s observe the output

|  |
| --- |
| /\* Output  In createDoubleVec  Calling jumpFunction with vec (lvalue).  **Copy Constructor**  In jumpFunction  **Copy Constructor**  Printing the vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  Calling jumpFunction with createDoubleVec (rvalue).  In createDoubleVec  In jumpFunction  **Copy Constructor**  Printing the vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  \*/ |

* As you see, when we invoke the jumpFunction with rvalue (jumpFunction(createDoubleVec(20))), copy constructor is getting called when it calls foo(arg).
* The solution is to have a T&& instead of T.

|  |
| --- |
| template<typename T>  void jumpFunction(T&& arg) {  *cout* << "In jumpFunction\n";  foo(*std*::*forward*<T>(arg));  } |

* The jumpFunction() takes T&& as argument and it will call the standard library function *std*::*forward* before passing the argument to function foo().
* Now let’s call the same code from main and observe the output.

|  |
| --- |
| /\* Output  In createDoubleVec  Calling jumpFunction with vec (lvalue).  In jumpFunction  Copy Constructor  Printing the vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  Calling jumpFunction with createDoubleVec (rvalue).  In createDoubleVec  In jumpFunction  Move Constructor  Printing the vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  \*/ |

* To perfect forwarding to work, C++11 has introduced the "reference collapsing" rule.
* This rule is applied by compiler during template argument deduction.
* You as a programmer cannot write a code as below, but complier can generate such code.
  1. T& & => T&
  2. T& && => T&
  3. T&& & => T&
  4. T&& && => T&&
* Example:
* If we call the function jumpFunction() as shown below...
  1. Case 1:
     + jumpFunction(9) -> here 9 is int rvalue reference which is int&&.
     + So, T will be replaced with int&&.
     + So, the jumpFunction(T&&) -> jumpFunction(int&& &&)
     + As per reference collapsing rule, this will become jumpFunction(int&&)
  2. Case 2:
     + int x;
     + jumpFunction(x) -> here x is lvalue.
     + Here T will be replaced with int&.
     + So, the jumpFunction(T&&) -> jumpFunction(int& &&)
     + As per reference collapsing rule, will become jumpFunction(int&)