Look at this piece of code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <iostream>  #include <functional>    using namespace std;  using namespace std::placeholders;    void add(int a, int b, int& r)  {      r = a + b;  }    int main()  {      int result = 0;        auto f = bind(add, \_1, 20, result);        f(80);        cout << result << endl;      return 0;  } |

This program supposedly adds 80 to 20 and prints the result; it compiles perfectly but when you execute it; you get…. **0**!

Why?

Because the bind method receives its parameters as parameters-by-value and the “result” variable is copied before being passed to the bound function add. Why?  
Because bind does not know if the parameters will still be valid when the actual invocation will be performed (remember, you could pass a functionobject to other function passing local variables as arguments and invoking it from there).

The solution? Pretty simple:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int main()  {      int result = 0;        auto f = bind(add, \_1, 20, ref(result));        f(80);        cout << result << endl;      return 0;  } |

I added the function ref that sends our parameter as a reference to the bound function.

What does this function ref do?

It is a template function that returns a reference\_wrapper object. A reference\_wrapper is a class template that wraps a reference in a concrete object.

Actually you could do something like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int main()  {      int result = 0;        reference\_wrapper<int> result\_ref(result);      auto f = bind(add, \_1, 20, result\_ref);        f(80);        cout << result << endl;      return 0;  } |

and everything would continue working as expected.

As you can see, you can pass the reference\_wrapper by value and everything will work because its copy constructor copies the reference (actually, the reference\_wrapper implementations do not store a reference but a pointer to the data being referenced, but their methods expose it as a reference).

Other nice usage of this would be in cases where you need to have a container of references (the actual objects are stored in other container or in other place and you do not need/want to have copies or pointers to them). For example, you have these classes:

|  |  |
| --- | --- |
| 1  2 | class A { };  class B : public A { }; |

And you want to have at the same time local variables pointing to them and you want them stored in a container:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int main()  {    A a, c;    B b, d;      vector<A> v = { a, b, c, d };  } |

Good? No! Bad at all! You are storing instances of class A in your vector. All the instances of B will be copied as instances of A (losing their specific attributes and all the polymorphic behavior and so on).

One solution? Storing pointers:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int main()  {    A a, c;    B b, d;      vector<A\*> v = { &a, &b, &c, &d };  } |

It works, but it is not evident for the user of the container if s/he will be in charge of freeing the objects or not.

Other solution? Using references:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int main()  {    A a, c;    B b, d;      vector<A&> v = { a, b, c, d };  } |

Looks nice, but it does not compile; because you cannot specify reference types in a vector.

Real solution: Using reference\_wrappers:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int main()  {    A a, c;    B b, d;      vector<reference\_wrapper<A>> v = { a, b, c, d };  } |

Someone could argue: In which scenario is this thing useful?

If you create a UI frame using Java Swing, you probably create a subclass of the JFrame class, will specify your visual components as member variables and you will also add them into the JFrame’s component list. Implementing something similar in C++ using reference\_wrappers would be quite elegant.