C++11右值引用

1.左值引用和右值引用

传统的C++语法中就有引用的语法，而C++11中新增了的右值引用语法特性，所以从现在开始我们之前学习的引用就叫做左值引用。**无论左值引用还是右值引用，都是给对象取别名。**

什么是左值？什么是左值引用？

左值是一个表示数据的表达式(如变量名或解引用的指针)，**我们可以获取它的地址+可以对它赋值，左值可以出现赋值符号的左边，右值不能出现在赋值符号左边。**定义时const修饰符后的左值，不能给他赋值，但是可以取它的地址。左值引用就是给左值的引用，给左值取别名。

```
void test() {
    // 以下的p、b、c、*p都是左值

    int *p = new int(0);
    int b = 1;
    const int c = 2;
    // 以下几个是对上面左值的左值引用
    int *&rp = p;
    int &rb = b;
    const int &rc = c;
    int &pvalue = *p;
}
```

什么是右值？什么是右值引用？

右值也是一个表示数据的表达式，如：`字面常量`、`表达式返回值`，`函数返回值`(这个不能是左值引用返回)等等，**右值可以出现在赋值符号的右边，但是不能出现出现在赋值符号的左边，右值不能取地址。**

右值引用是C++11引入的一种引用类型，用于绑定到右值。它通过使用双引号&&来声明。右值引用可以将其绑定到一个右值，允许对其进行`移动语义`和`完美转发`。

```
void test() {
    double x = 1.1, y = 2.2;
    // 以下几个都是常见的右值
    10;
    x + y;
    fmin(x, y);
    // 以下几个都是对右值的右值引用
    int &&rr1 = 10;
    double &&rr2 = x + y;
    double &&rr3 = fmin(x, y);
    // 这里编译会报错：error C2106: “=”: 左操作数必须为左值
    10 = 1;
    x + y = 1;
    fmin(x, y) = 1;
}
```

需要注意的是右值是不能取地址的，但是给右值取别名后，会导致右值被存储到特定位置，且可以取到该位置的地址，也就是说例如：不能取字面量10的地址，但是rr1引用后，可以对rr1取地址，也可以修改rr1。如果不想rr1被修改，可以用const int&& rr1去引用

```
void test() {
    double x = 1.1, y = 2.2;
    int &&rr1 = 10;
    const double &&rr2 = x + y;
    rr1 = 20;
    rr2 = 5.5;// 报错
    return 0;
}
```

const左值引用右值

在 C++ 中，`const` 左值引用可以绑定到右值，但是需要一些特定的条件。在 C++11 引入了右值引用和移动语义之后，可以将右值绑定到左值引用的 `const` 版本上。

当将右值绑定到 `const` 左值引用时，编译器会创建一个临时对象，该对象是右值的副本，并且该临时对象的生命周期与左值引用的生命周期相同。这样，通过 `const` 左值引用，可以安全地访问右值的内容。

```
void test() {
    int x = 10;
    cout << x << endl;   // 正确，x 是左值
    cout << 20 << endl;  // 正确，20 是右值
    const int &ref = x;  // 正确，将左值绑定到 const 左值引用
    const int &temp = 30;// 正确，将右值绑定到 const 左值引用，创建一个临时对象
}
```

> `const` 左值引用可以绑定到右值，但是会创建一个临时对象来容纳右值的副本。这种绑定方式可以安全地访问右值的内容，并且在某些情况下可以提供更高的灵活性和代码复用性。

2.左值引用和右值引用比较

左值引用总结：

> 1. 左值引用只能引用左值，不能引用右值。
> 2. 但是const左值引用既可引用左值，也可引用右值。

```
void test() {
    // 左值引用只能引用左值，不能引用右值。
    int a = 10;
    int &ra1 = a; // ra为a的别名
    int &ra2 = 10;// 编译失败，因为10是右值
    // const左值引用既可引用左值，也可引用右值。
    // 编译器会生成一个临时的常量对象，并将const左值引用绑定到该临时对象上。
    const int &ra3 = 10;
    const int &ra4 = a;
}
```

> **const左值引用右值，编译器会生成一个临时的常量对象，并将const左值引用绑定到该临时对象上。**

右值引用总结：

> 1. 右值引用只能右值，不能引用左值。
> 2. 但是右值引用可以引用move之后的左值。

```
void test() {
    // 右值引用只能右值，不能引用左值。
    int &&r1 = 10;

    // message : 无法将左值绑定到右值引用
    int a = 10;
    int &&r2 = a;   // error C2440: “初始化”: 无法从“int”转换为“int &&”
    // 右值引用可以引用move以后的左值
    int &&r3 = std::move(a);
}
```

3.move

`std::move` 是 C++ 标准库中的一个函数模板，位于 `<utility>` 头文件中。它用于将对象的所有权从一个对象转移到另一个对象，通常用于实现移动语义和避免不必要的对象拷贝操作。

`std::move` 的函数原型如下：

```
template<class T>
constexpr typename std::remove_reference<T>::type&& move(T&& t) noexcept;
```

该函数接受一个对象 `t`，并将其转换为右值引用，返回一个指向转换后的右值引用的对象。右值引用表示对象的所有权可以被移动或转移，而不是进行拷贝。`std::move` 本质上是**将左值强制转换为右值引用，从而告诉编译器该对象可以被移动而非拷贝。**

使用 `std::move` 的主要用途是在实现移动语义时，将对象的资源转移给其他对象，以提高效率。移动语义允许在对象所有权的转移过程中，将资源（如动态分配的内存或打开的文件句柄）从一个对象转移到另一个对象，而不进行不必要的拷贝操作。

下面是一个简单的示例，演示了 `std::move` 的用法：

```
#include <iostream>
#include <utility>

class MyClass {
public:
    MyClass() {
        std::cout << "无构造" << std::endl;
    }

    MyClass(const MyClass& other) {
        std::cout << "拷贝构造" << std::endl;
    }
```

```
    MyClass(MyClass&& other) noexcept {
        std::cout << "移动构造" << std::endl;
    }
};

int main() {
    MyClass obj1;
    MyClass obj2 = std::move(obj1);  // 调用移动构造函数
    return 0;
}
```

在上面的示例中，`obj1` 和 `obj2` 都是 `MyClass` 类型的对象。通过调用 `std::move(obj1)`，我们将 `obj1` 的所有权转移给了 `obj2`，因此在转移过程中会调用移动构造函数。这样做可以避免调用拷贝构造函数，提高了程序的效率。

需要注意的是，使用 `std::move` 之后，原对象的状态是不确定的，它可能处于有效状态、空状态或不可用状态。因此，在使用 `std::move` 之后，对原对象的操作应该谨慎，通常应该避免使用原对象。

> `std::move` 是 C++ 中用于转移对象所有权的函数模板。它将左值转换为右值引用，从而告诉编译器该对象可以进行移动操作。通过使用 `std::move`，可以实现移动语义，避免不必要的对象拷贝，提高程序的效率。

4.右值引用使用场景和意义

前面我们可以看到左值引用既可以引用左值和又可以引用右值，那为什么C++11还要提出右值引用呢？是不是化蛇添足呢？下面我们来看看左值引用的短板，右值引用是如何补齐这个短板的！

下面是string类的实现

```
namespace phw {
    class string {
    public:
        typedef char *iterator;
        iterator begin() {
            return _str;
        }
        iterator end() {
            return _str + _size;
        }
        string(const char *str = "")
            : _size(strlen(str)), _capacity(_size) {
            //cout << "string(char* str)" << endl;
            _str = new char[_capacity + 1];
            strcpy(_str, str);
        }
        // s1.swap(s2)
        void swap(string &s) {
            ::swap(_str, s._str);
            ::swap(_size, s._size);
            ::swap(_capacity, s._capacity);
        }
        // 拷贝构造
        string(const string &s)
```

```
        : _str(nullptr) {
        cout << "string(const string& s) -- 深拷贝" << endl;
        string tmp(s._str);
        swap(tmp);
    }
    // 赋值重载
    string &operator=(const string &s) {
        cout << "string& operator=(string s) -- 深拷贝" << endl;
        string tmp(s);
        swap(tmp);
        return *this;
    }
    // 移动构造
    string(string &&s)
        : _str(nullptr), _size(0), _capacity(0) {
        cout << "string(string&& s) -- 移动语义" << endl;
        swap(s);
    }
    // 移动赋值
    string &operator=(string &&s) {
        cout << "string& operator=(string&& s) -- 移动语义" << endl;
        swap(s);
        return *this;
    }
    ~string() {
        delete[] _str;
        _str = nullptr;
    }
    char &operator[](size_t pos) {
        assert(pos < _size);
        return _str[pos];
    }
    void reserve(size_t n) {
        if (n > _capacity) {
            char *tmp = new char[n + 1];
            strcpy(tmp, _str);
            delete[] _str;
            _str = tmp;
            _capacity = n;
        }
    }
    void push_back(char ch) {
        if (_size >= _capacity) {
            size_t newcapacity = _capacity == 0 ? 4 : _capacity * 2;
            reserve(newcapacity);
        }
        _str[_size] = ch;
        ++_size;
        _str[_size] = '\0';
    }
    //string operator+=(char ch)
    string &operator+=(char ch) {
        push_back(ch);
        return *this;
    }
```

```
        const char *c_str() const {
            return _str;
        }

    private:
        char *_str;
        size_t _size;
        size_t _capacity;// 不包含最后做标识的\0
    };
}// namespace phw
```

左值引用的使用场景

做参数和做返回值都可以提高效率。

```
void func1(phw::string s)
{}

void func2(const phw::string& s)
{}

int main()
{
    phw::string s1("hello world");
    // func1和func2的调用我们可以看到左值引用做参数减少了拷贝，提高效率的使用场景和价值
    func1(s1);   //深拷贝
    func2(s1);   //没有深拷贝
    //下面是string类中+=的定义，一个是传值，一个是传引用
    //string operator+=(char ch);  //传值返回存在深拷贝
    //string& operator+=(char ch);  //传左值引用没有拷贝提高了效率
    s1 += '!';
    return 0;
}
```

左值引用的短板

当函数返回对象是一个局部变量，出了函数作用域就不存在了，就不能使用左值引用返回， 只能传值返回。例如：string to_string(int value)函数中可以看到，这里只能使用传值返回， 传值返回会导致至少1次拷贝构造(如果是一些旧一点的编译器可能是两次拷贝构造)。

```
string to_string(int value) {
    bool flag = true;
    if (value < 0) {
        flag = false;
        value = 0 - value;
    }
    string str;
    while (value > 0) {
        int x = value % 10;
        value /= 10;
        str += ('0' + x);
    }
    if (flag == false) {
```

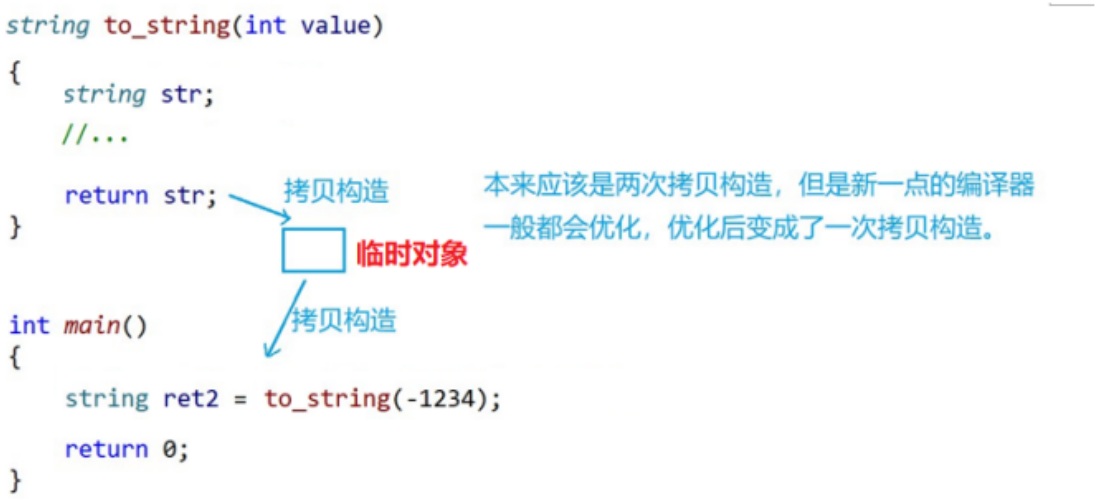
```
        str += '-';
    }

    std::reverse(str.begin(), str.end());
    return str;
}
// str是一个局部对象，只能传值返回，因为局部对象出了函数外就销毁了，不能引用
```

这个函数的返回值是一个右值。在C++中，当一个函数返回一个临时对象时，该对象被视为右值。

对于该函数的返回值，它是一个临时创建的`string`对象，它的生命周期仅限于函数返回后的瞬间。由于这个临时对象没有持久性，编译器可以对其进行优化，例如通过将返回值直接移动（move）到调用者的位置，而不是执行复制操作。

根据C++标准库的规范，标准函数`std::reverse`不会改变迭代器的有效性，所以在调用`std::reverse`后，`str`仍然是一个有效的右值。因此，编译器可能会继续优化，例如对`str`进行移动操作，而不是执行复制操作。

```
int main()
{
    string ret1 = to_string(1234);
    string ret2 = to_string(-1234);
    return 0;
}
```



```cpp
// 拷贝构造
string(const string& s)
    :_str(nullptr)
{
    cout << "string(const string& s) -- 深拷贝" << endl;

    string tmp(s._str);
    swap(tmp);
}
string to_string(int value)
{
    string str;
    //...

    return str;
}

int main()
{
    string ret2 = to_string(-1234);

    return 0;
}
```

函数返回一个临时对象时，该对象被视为右值

to_string的返回值是一个右值，用这个右值构造ret2，如果没有移动构造，调用就会匹配调用拷贝构造，因为const左值引用是可以引用右值的，这里就是一个深拷贝。

string to_string(int value)函数中可以看到，这里只能使用传值返回，传值返回会导致至少1次拷贝构造(如果是一些旧一点的编译器可能是两次拷贝构造)。使用移动构造可以解决多次拷贝构造的问题。

5.移动构造和移动语义

移动构造（move constructor）是一种特殊的构造函数，用于从一个对象移动（或者说窃取）资源而不是复制资源。移动构造函数通常采用右值引用作为参数，并将资源从传入的对象转移到正在构造的对象中。移动构造函数可以通过使用移动语义来提高性能，因为它可以避免昂贵的复制操作。

移动语义（move semantics）是一种语言特性，允许在适当的情况下将资源从一个对象移动到另一个对象，而不是进行复制操作。移动语义的实现依赖于移动构造函数和移动赋值运算符。

在string中增加移动构造，**移动构造本质是将参数右值的资源窃取过来，占位已有，那么就不用做深拷贝了，所以它叫做移动构造，就是窃取别人的资源来构造自己。**

```cpp
// 拷贝构造
string(const string& s)
    : _str(nullptr) {
    cout << "string(const string& s) -- 深拷贝" << endl;
    string tmp(s._str);
    swap(tmp);
}

// 移动构造
string(string &&s)
    : _str(nullptr), _size(0), _capacity(0) {
    cout << "string(string&& s) -- 移动语义" << endl;
```

```
    swap(s);
}


int main() {
    string ret2 = to_string(-1234);
    return 0;
}
```

再运行上面to_string的两个调用，我们会发现，这里没有调用深拷贝的拷贝构造，而是调用了移动构造，移动构造中没有新开空间，拷贝数据，所以效率提高了。

> to_string的返回值是一个右值，用这个右值构造ret2，如果既有拷贝构造又有移动构造，调用会匹配调用移动构造，因为编译器会选择最匹配的参数调用。那么这里就是一个移动语义。

6.移动赋值

在string类中增加移动赋值函数，再去调用to_string(1234)，不过这次是将 to_string(1234)返回的右值对象赋值给ret1对象，这时调用的是移动构造。

```
// 移动赋值
string &operator=(string &&s) {
    cout << "string& operator=(string&& s) -- 移动语义" << endl;
    swap(s);
    return *this;
}

int main() {
    string ret1;
    ret1 = to_string(1234);
    return 0;
}
// 运行结果:
// string(string&& s) -- 移动语义
// string& operator=(string&& s) -- 移动语义
```

这里运行后，我们看到调用了一次移动构造和一次移动赋值。因为如果是用一个已经存在的对象接收，编译器就没办法优化了。to_string函数中会先用str生成构造生成一个临时对象，但是我们可以看到，编译器很聪明的在这里把str识别成了右值，调用了移动构造。然后在把这个临时对象做为to_string函数调用的返回值赋值给ret1，这里调用的移动赋值。

> 在C++11中STL中的容器都是增加了移动构造和移动赋值

7.右值引用引用左值及其一些更深入的使用场景分析

按照语法，右值引用只能引用右值，但右值引用一定不能引用左值吗？因为：有些场景下，可能真的需要用右值去引用左值实现移动语义。**当需要用右值引用引用一个左值时，可以通过move函数将左值转化为右值。**C++11中，`std::move()`函数位于头文件`<utility>`中，该函数名字具有迷惑性， 它并不搬移任何东西，**唯一的功能就是将一个左值强制转化为右值引用，然后实现移动语义。**

左值引用没解决的问题：

- 局部对象返回问题(局部对象出了作用域被销毁)
- 插入接口，对象拷贝问题(传参，会发生拷贝)

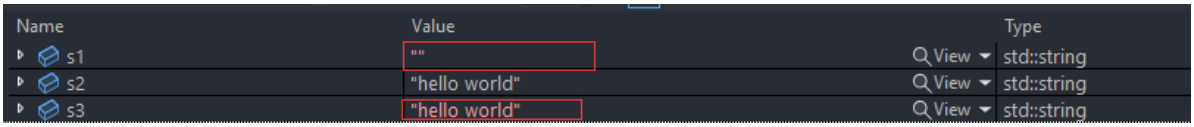
下面看个例子：

```
template<class T>
T func(){
    T ret;

    //...
    return ret;
}

T x  = Func();
```

T是一个自定义类型：

> 1、如果T是浅拷贝的类，这里就是拷贝构造，因为对于浅拷贝的类，移动构造是没什么意义的。
>
> 2、如是T是深拷贝的类，这里就是移动构造，对于深拷贝，移动构造可以转移右值的资源，没有拷贝

```
int main() {
    string s1("hello world");
    // 这里s1是左值，调用的是拷贝构造
    string s2(s1);
    // 这里我们把s1 move处理以后，会被当成右值，调用移动构造
    // 但是这里要注意，一般是不要这样用的，因为我们会发现s1的资源被转移给了s3，s1被置空了。
    string s3(std::move(s1));
    return 0;
}
```



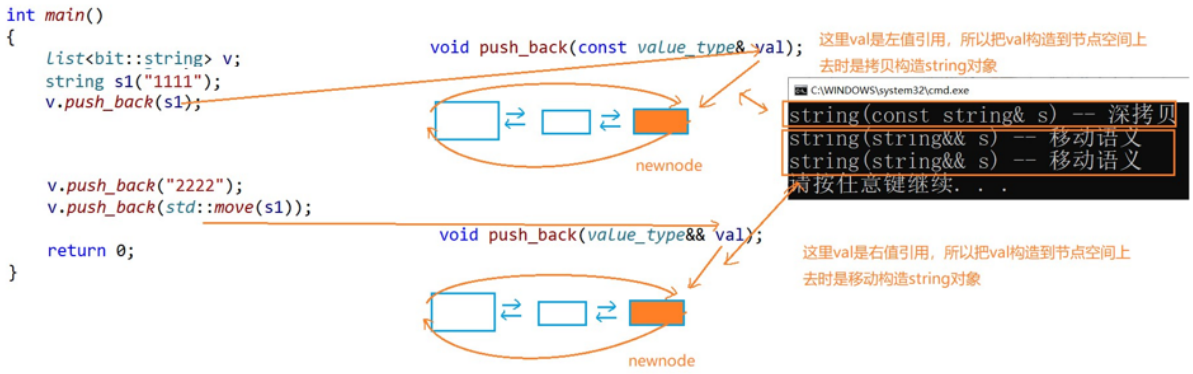
> STL容器插入接口函数也增加了右值引用版本：

```
void push_back(value_type &&val);
int main() {
    list<string> lt;
    string s1("1111");
    // 这里调用的是拷贝构造
    lt.push_back(s1);
    // 下面调用都是移动构造
    lt.push_back("2222");
    lt.push_back(std::move(s1));
    return 0;
}
```

运行结果：

```
// string(const string& s) -- 深拷贝
// string(string&& s) -- 移动语义
```

```
// string(string&& s) -- 移动语义
void Fun(int &x) { cout << "左值引用" << endl; }
void Fun(const int &x) { cout << "const 左值引用" << endl; }
```



8.完美转发

完美转发（perfect forwarding）是一种技术，允许函数模板将其参数传递给其他函数，并保留原始参数的值类别（左值或者右值）。这是在C++11中引入的新功能，通过引入两个新的引用限定符 `&&` 来实现。

完美转发的主要目的是解决函数模板中参数传递的问题。通常情况下，当我们将一个参数传递给函数模板的另一个函数时，参数的值类别会发生改变，比如一个右值可能会被转换为左值。这可能会导致一些问题，特别是在涉及重载和模板的情况下。

通过使用完美转发，我们可以确保参数的值类别保持不变。下面是一个示例代码，展示了如何使用完美转发：

```
#include <utility>

// 接受任意参数类型的函数模板
template <typename T>
void forwardFunction(T&& arg){
    otherFunction(std::forward<T>(arg)); // 完美转发参数
}

// 接受一个左值引用的函数
void otherFunction(int& arg){
    // 处理左值引用
}

// 接受一个右值引用的函数
void otherFunction(int&& arg){
    // 处理右值引用
}

void otherFunction(const int& arg){
    // 处理const左值引用
}

void otherFunction(const int&& arg){
    // 处理const右值引用
}
```

```
int main(){
    int value = 42;
    const int value2 = 10;
    forwardFunction(value);         // 传递左值
    forwardFunction(123);           // 传递右值
    forwardFunction(value2);        // 传递const左值
    forwardFunction(std::move(value2));    //传递const右值

    return 0;
}
```

在上面的示例中，`forwardFunction`是一个接受任意类型参数的函数模板。通过使用`T&&`作为参数类型，我们实现了完美转发。然后，通过`std::forward<T>(arg)`来传递参数给`otherFunction`，确保参数的值类别保持不变。

`otherFunction`有四个重载版本，分别接受左值引用和右值引用以及对应的const版本。在`forwardFunction`中，我们可以传递左值`value`和右值`123`，也可以传递const左值，它们分别将被正确地转发到相应的`otherFunction`。

> - 模板中的&&不代表右值引用，而是万能引用，其既能接收左值又能接收右值。
> - 模板的万能引用只是提供了能够接收同时接收左值引用和右值引用的能力，
> - 但是引用类型的唯一作用就是限制了接收的类型，后续使用中都退化成了左值，
> - 我们希望能够在传递过程中保持它的左值或者右值的属性

`std::forward` 完美转发在传参的过程中保留对象原生类型属性

```
void Fun(int &x) { cout << "左值引用" << endl; }
void Fun(const int &x) { cout << "const 左值引用" << endl; }
void Fun(int &&x) { cout << "右值引用" << endl; }
void Fun(const int &&x) { cout << "const 右值引用" << endl; }
// std::forward<T>(t)在传参的过程中保持了t的原生类型属性。
template<typename T>
void PerfectForward(T &&t) {
    Fun(std::forward<T>(t));
}

int main() {
    PerfectForward(10);// 右值
    int a;
    PerfectForward(a);            // 左值
    PerfectForward(std::move(a));// 右值
    const int b = 8;
    PerfectForward(b);            // const 左值
    PerfectForward(std::move(b));// const 右值
    return 0;
}
```

完美转发实际中的使用场景：

```
template<class T>
```

```cpp
struct ListNode {
    ListNode *_next = nullptr;
    ListNode *_prev = nullptr;
    T _data;
};

template<class T>
class List {
    typedef ListNode<T> Node;

public:
    List() {
        _head = new Node;
        _head->_next = _head;
        _head->_prev = _head;
    }

    void PushBack(T &&x) {
        //Insert(_head, x);
        Insert(_head, std::forward<T>(x));
    }

    void PushFront(T &&x) {
        //Insert(_head->_next, x);
        Insert(_head->_next, std::forward<T>(x));
    }

    void Insert(Node *pos, T &&x) {
        Node *prev = pos->_prev;
        Node *newnode = new Node;
        newnode->_data = std::forward<T>(x);// 关键位置
        // prev newnode pos
        prev->_next = newnode;
        newnode->_prev = prev;
        newnode->_next = pos;
        pos->_prev = newnode;
    }

    void Insert(Node *pos, const T &x) {
        Node *prev = pos->_prev;
        Node *newnode = new Node;
        newnode->_data = x;// 关键位置
        // prev newnode pos
        prev->_next = newnode;
        newnode->_prev = prev;
        newnode->_next = pos;
        pos->_prev = newnode;
    }

private:
    Node *_head;
};

int main() {
    List<bit::string> lt;
```

```
    lt.PushBack("1111");
    lt.PushFront("2222");
    return 0;
}
```

完美转发被用于将参数从 `PushBack` 和 `PushFront` 函数传递到 `Insert` 函数中。

在List类的定义中，`PushBack` 和 `PushFront` 成员函数都接受右值引用参数 `T&& x`。当调用 `PushBack("1111")` 和 `PushFront("2222")` 时，字符串字面值被转换为右值引用。

然后，在 `PushBack` 和 `PushFront` 函数中，`Insert` 函数被调用，参数 `x` 被通过 `std::forward<T>(x)` 完美转发到 `Insert` 函数中。这里使用 `std::forward` 来确保参数的值类别（左值或者右值）保持不变。

在 `Insert` 函数中，`x` 被再次通过 `std::forward<T>(x)` 完美转发给新创建的 `Node` 对象的 `_data` 成员。这样做可以避免不必要的复制或移动，并保持原始参数的值类别。

通过使用完美转发，程序可以正确地将参数传递给 `Insert` 函数，并保持参数的值类别不变，从而避免不必要的拷贝和移动操作。这种方式提高了代码的效率，并允许程序处理不同类型的参数（左值或者右值）。