Object-Serialization 大作业报告

刘轩铭、蔡灿宇

实现内容与程序亮点

- ☑ 实现了基本要求（命名空间，STL类型，用户自定义等）
- ☑ 利用变长参数模板实现用户自定义类型的序列化
- ☑ vector容器支持嵌套，如vector<pair<int, double>>等嵌套定义和序列化
- ☑ 完成了Bonus：base64编码

Binary serialization/deserialization

- **API**

```
//serialize
void serialize(T object, std::string filename);         //arithmetic
void serialize(string str, std::string filename);       //string
void serialize(vector<T>& v, std::string filename);     //vector
void serialize(list<T>& l, std::string filename);       //list
void serialize(set<T>& s, std::string filename);        //set
void serialize(map<K, V>& m, std::string filename);     //map
//deserialize
void deserialize(T &object, std::string filename);      //arithmetic
void deserialize(string &str, std::string filename);    //string
void deserialize(vector<T>& v, std::string filename);   //vector
void deserialize(list<T>& l, std::string filename);     //list
void deserialize(set<T>& s, std::string filename);      //set
void deserialize(map<K, V>& m, std::string filename);   //map
//userDefined
void userDefinedType(string file, H head, Args ...rest);
```

- **运行机制**
  - 调用serialize和deserialize函数，将不同类型的参数传递进去。
  - 根据不同的参数跳转到不同的重载函数，进行分别二进制文件的读写或访问。
  - 最后将serialize和deserialize的结果进行比对，输出结果。
- **用户自定义**

  1. 使用userDefinedType函数

```
void userDefinedType(string file)
    return;
template<class H,class ...Args>
void userDefinedType(string file, H head, Args ...rest)
{
    serialize(head, file);
    userDefinedType(file, rest...);
}
```

具体实现如下：

```
struct UserDefinedType testStru = { 5, "Tom", {1.2, 3.4, 7, 7} };
userDefinedType("n.data", testStru.idx, testStru.name, testStru.data);
```

将用户自定义结构里的类型分别作为可变参数传入函数，实现用户自定义类型的序列化。

2. 将各个内容进行序列化

具体实现如下：

```
struct UserDefinedType u = { 5, "Tom", {1.2, 3.4, 7, 7} };
struct UserDefinedType v;
serialize(u.idx, "idx.data");
serialize(u.name, "name.data");
serialize(u.data, "data.data");
deserialize(v.idx, "idx.data");
deserialize(v.name, "name.data");
deserialize(v.data, "data.data");
if ((u.idx == v.idx)&& (u.name == v.name)&& (u.data == v.data))
    cout << "UserDefinedType is current!" << endl;
else cout << "UserDefinedType is wrong!" << endl;
```

将定义的结构内的类型分别进行序列化和反序列化，得到想要的结果。

- **测试数据**

```
void arithmetic_type_1();   //int
void arithmetic_type_2();   //double
void string_type();         //string
void pair_type_1();         //pair<int, double>
void vector_type_1();       //vector<int>
void vector_type_2();       //vector<double>
void vector_type_3();       //vector<string>
void vector_type_4();       //vector<pair<int, double>>
void list_type_1();         //list<int>
void list_type_2();         //list<double>
void list_type_3();         //list<string>
void set_type_1();          //set<int>
void set_type_2();          //set<double>
void set_type_3();          //set<string>
void map_type_1();          //map<int, string>
void map_type_2();          //map<double, string>
void map_type_3();          //map<int, double>
```

将常见的类型都包含了进去，实现了vectot的嵌套vector<pair<int, double>>

- **测试结果**

全部通过，详情可以运行工程文件查看。

XML serialization/deserialization

- **API**

```
void serialize_xml(int& obj, string type, string file_name);
```

```
void serialize_xml(short& obj, string type, string file_name);
void serialize_xml(double& obj, string type, string file_name);
void serialize_xml(float& obj, string type, string file_name);
void serialize_xml(long& obj, string type, string file_name);
void serialize_xml(bool& obj, string type, string file_name);
void serialize_xml(long long& obj, string type, string file_name);
void serialize_xml(long double& obj, string type, string file_name) ;
void serialize_xml(char& obj, string type, string file_name);
void serialize_xml(string& obj, string type, string file_name);
void serialize_xml(wchar_t& obj, string type, string file_name);
void serialize_xml(const std::vector<Vec>& obj, string type, string
file_name);
void serialize_xml(const std::pair<First, Second>& obj, string type, string
file_name);
void serialize_xml(const std::list<List>& obj, string type, string
file_name);
void serialize_xml(const std::set<Set>& obj, string type, string file_name);
void serialize_xml(const std::map<First, Second>& obj, string type, string
file_name);

void deserialize_xml(int& obj, string type, string file_name);
void deserialize_xml(short& obj, string type, string file_name);
void deserialize_xml(float& obj, string type, string file_name);
void deserialize_xml(double& obj, string type, string file_name);
void deserialize_xml(bool& obj, string type, string file_name);
void deserialize_xml(long double& obj, string type, string file_name);
void deserialize_xml(long long& obj, string type, string file_name);
void deserialize_xml(long& obj, string type, string file_name);
void deserialize_xml(char& obj, string type, string file_name);
void deserialize_xml(wchar_t& obj, string type, string file_name);
void deserialize_xml(string& obj, string type, string file_name);
void deserialize_xml(std::vector<Vec>& obj, string type, string file_name);
void deserialize_xml(std::pair<First, Second>& obj, string type, string
file_name);
void deserialize_xml(std::list<List>& obj, string type, string file_name);
void deserialize_xml(std::set<Set>& obj, string type, string file_name);
void deserialize_xml(std::map<First, Second>& obj, string type, string
file_name);

void userDefinedTypeToXML (string file, H head, Args... rest);
```

- **运行机制**
  - 调用XMLSerializerBase这个类，生成一个对应类型的对象。
  - 对这个对象调用serialize和deserialize函数。
  - 根据传入的不同参数跳转到不同的重载函数上实现xml的序列化或反序列化。
  - 将结果输出到.xml文件中。
- **用户自定义**

  使用userDefinedTypeToXML函数接口进行用户自定义类型的序列化实现。

```
template <class ...Args>
void userDefinedTypeToXML (string file, Args... args) {
    reset();
    userDefinedType2XML(file, args...);
}
template <class H, class ...Args>
void userDefinedType2XML (string file, H head, Args... rest) {
    serialize(head, root);
    userDefinedType2XML(file, rest...);
}
void userDefinedType2XML (string file)
    xml_save(file);
```

具体实现如下：

```
struct UserDefinedType testStru = { 5, "Tom", {1.2, 3.4, 7, 7} };
XMLSerializerBase<int> base;
base.userDefinedTypeToXML("testStruct.xml", testStru.idx, testStru.name,
testStru.data);
```

这里采用了函数变长参数模板，将用户自定义结构里的类型分别作为可变参数传入函数，实现用户自定义类型的序列化。

- **测试数据**

```
void xml_arithmetic_1();            // int
void xml_arithmetic_2();            // double
void xml_arithmetic_3();            // char
void xml_string();                  // string
void xml_vector_int();              // vector<int>
void xml_vector_vector_int();       // vector<vector<int>>
void xml_vector_string();           // vector<string>
void xml_vector_pair();             // vector<pair<int, double>>
void xml_list_int();                // list<int>
void xml_pair_1();                  // pair<int, float>
void xml_pair_2();                  // pair<vector<int>, list<float>>
void xml_set_string();              // set<string>
void xml_map_1();                   // map<double, string>
void xml_map_2();                   // map<vector<int>, list<float>>
```

将常见类型全部包含，实现了各种类型的嵌套

- **测试结果**

全部通过，详情可运行工程文件查看。

base64 Bonus的实现

- **解决思路**

将XML文件每三个字节进行读出，然后以6个Bit作为单位，进行切分，高位补足两个0。这样可以将XML三个字节扩展成四字节形式，且每个字节代表的数范围为0-63.

根据Base64编码规则，如下：

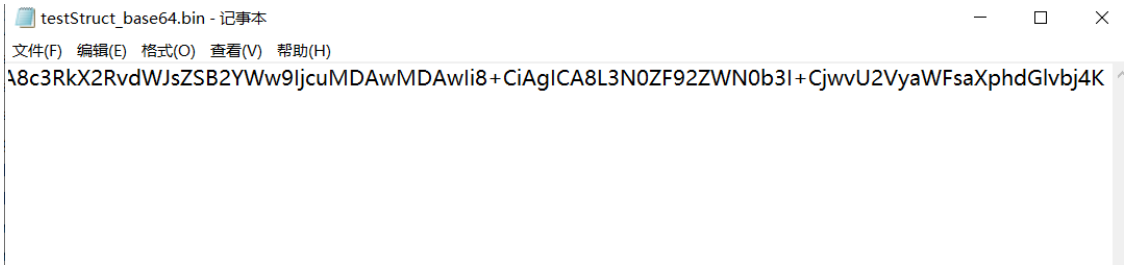
索引	对应字符	索引	对应字符	索引	对应字符	索引	对应字符
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

进行相应的映射，可以对每个新得到的字节进行编码，得到相应的四位Base64编码。

- **代码呈现**

```
 void XMLSerializerBase<T>::xml2Base64(string name)
    {
        FILE* fp_xml = fopen(name.c_str(), "r");
        FILE* fp_base64 = fopen((name.substr(0, name.find('.')) +
"_base64.bin").c_str(), "w+");
        if((!fp_xml) || (!fp_base64)) return;
        char tmp[3], base64tmp[4], base64[4];
        int cnt = 0, numRead;
        for( numRead = fread(tmp, 1, 3, fp_xml); numRead != 0 &&
(!feof(fp_xml)) ; fread(tmp, 1, 3, fp_xml)) {
            if(numRead == 2)
                tmp[2] = 0;
            else if(numRead == 1)
                tmp[2] = tmp[1] = 0;
            base64tmp[0] = (tmp[0] >> 2) & 0x3f;
            base64tmp[1] = (((tmp[0] & 0x03) << 4) | (tmp[1] >> 4)) & 0x3f;
            base64tmp[2] = (((tmp[1] & 0x0f) << 2) | (tmp[2] >> 6)) & 0x3f;
      base64tmp[3] = (tmp[2] & 0x3f) & 0x3f;
      for (int i = 0; i < 4; i++) {
        if (base64tmp[i] >= 0 && base64tmp[i] < 26)
          base64[i] = base64tmp[i] + 65;
        else if (26 <= base64tmp[i] && base64tmp[i] < 52)
          base64[i] = base64tmp[i] + 71;
        else if (52 <= base64tmp[i] && base64tmp[i] < 62)
          base64[i] = base64tmp[i] - 4;
        else if (base64tmp[i] == 62)
          base64[i] = 43;
        else base64[i] = 47;
      }
            fwrite(base64, 1, 4, fp_base64);
        }
    }
```

- **运行结果**



- **结果验证**

用网上的编解码器进行验证，发现结果正确。详情可以参见工程文件。

程序分工

- **刘轩铭**：XML module + Base64编码实现 + 程序整合
- **蔡灿宇**：Binary module + 报告撰写