**2011软件构造试卷及答案**作者：[沈扬勇](http://blog.renren.com/GetEntry.do?id=854555592&owner=339815559)

考试科目名称　  软件构造

考试方式：开卷　　　　考试日期　2011　　年　6 月　25日　教师　郑滔

系（专业）　软件学院　　　　　　　年级　　2　　　　　班级

学号　　　　　　　　　　　　　姓名　　　　　　　　　　　成绩

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 题号 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 |
| 分数 |  |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| 得分 |  |

1、  请阐述什么是软件构造，以及软件构造过程中涵盖的activities。（本题满分　15　分）

**参考答案：**

按照一般的用法，“构造”是指建设的过程，构造过程可能包含有计划、设计、检查工作的一些方面，但在多数时候，“构造”就是指创建事物过程中动手的那些部分。“软件构造”就是创建软件过程中动手的那些部分。

开发计算机软件是一个复杂的过程。研究者认识到在软件开发过程中的活动有：

定义问题、需求分析、规划构建、软件架构（或高层设计）、详细设计、编码与调试、单元测试、集成测试、集成、系统测试、保障维护

其中规划构建、详细设计、编码与调试、单元测试、集成与集成测试属于软件构造过程。

评：题目很简单，课本上有直接答案；但是要注意软件开发过程中的活动与软件构造过程的活动的区别。不仔细的同学会把所有过程抄上；这样的应适当扣分

|  |  |
| --- | --- |
| 得分 |  |

2. 在设计某个系统的时候，使用了层次体系结构模式。其中第N层使用了第N-1层的服务。分析的时候发现：第N-1层的接口中包括了多个类：ClassInterface1, ClassInterface2, …, ClassInterfacen。并且市场上有多种对于第N-1层的实现。每个实现都将实现同样种类的类。为了保证第N-1层可以使用不同的实现，而不影响第N层的设计和代码，应该使用那种设计模式？给出设计的简单说明。（本题满分　15　分）

**参考答案：**

在这里，我们应该选择**桥接模式**。

**理由如下**：

l  我们不希望抽象和它的具体实现部分之间有一个固定的绑定关系，例如，有时我们可能需要在程序运行时刻，实现部分应可以被选择或者切换

l  类的抽象以及它的实现部分都应该可以通过生成子类的方法加以扩充，这是，这时桥接模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充

l  对一个抽象的实现部分的修改应该对客户不产生影响，即客户的代码不必重新编译

l  嵌套普化

l  分离接口及其实现部分，提高可扩充性，并能实现细节对客户透明

**示意图如下**：

**本题注意点**：

1、  如果采用桥接模式，只需要画出一到两个ClassInterface及其Impementor之间的关系图即可。

2、  如果采用其它设计模式，只要设计合理，有理论可行性，言之有理，即可，根据具体情况，酌情给分，不限于桥接模式

3、  其它可行的设计模式包括抽象工厂等

|  |  |
| --- | --- |
| 得分 |  |

3、（本题满分 30分）

在云计算兴起的今天，在某市需要建立一个云计算中心，该中心拥有过10万台的计算机，分别放置于1000间机房，可以为用户提供IAAS、PAAS和SAAS的服务。为了良好的管理这些计算机，需要开发一个管理系统CMIS，对计算机的工作环境进行监控。每个机房将安置一个关于温度、湿度和气压的感知装置M，该装置M每10秒钟采集一次数据，并存储于装置M中的文件F中。

要求：

CMIS每10分钟借助网络，分别从1000间机房的装置中，通过协议A以串口通讯方式远程读取装置M中的文件F，采集温度、湿度和气压数据，存放在内存中（链表等动态数据管理方式组织），对于异常情况及时报警，每小时生成一份统计数据，并将原始数据和统计数据存入数据库。

* 请给出CMIS的设计，包括子系统的划分、类的设计（以伪代码表示，注意伪码书写原则），设计中应尽量提高系统的可扩展性，例如：感知装置的变化（通讯方式、协议、数据格式等）
* 如何以防御式编程思想降低动态内存数据管理可能存在的隐患（内存泄漏、悬挂指针等）？请给出解决思路。

**参考答案：**

1、系统应该包括信息采集模块、数据内存存储管理以数据处理和预警、数据存储模块。信息采集模块应该能够将通讯、数据的收集、网络连接管理（包括建立连接和释放连接等）模块所应具有的功能抽取出接口，而具体的实现可以使用桥接模式或者策略模式去实现。数据内存存储管理以数据处理和预警模块应该首先考虑到数据节点的个数，内存的容量，是否需要对内存进行管理，需要考虑将来机房扩容后，内存容量的极限问题，需要适当考虑在内存和磁盘之间数据的调度和加载。数据存储模块中需要考虑数据的格式和批量数据处理的效率，如果是数据库，应考虑批处理等问题。在这几个模块的交互过程中，要考虑其间数据的交互，建议定义一种模块内部进行数据交换的数据格式，而在各模块提供其对应数据格式与该通用数据格式的转换接口，进而适应将来可能变化的外部提取的数据格式。同时，如若考虑负载均衡，则应该鼓励。

具体可选的参考答案如下,但在上述原则下，合理可行即可:

信息传输模块，负责将装置M采集到的信息采集并传输至信息中心

信息中心模块 负责将收集的信息处理，并生成统计数据

数据库管理模块 负责数据和数据库系统的交互

异常管理模块 负责处理系统异常的情况

类设计，系统主要需要以下几个类

AbstractComputerRoomData房间信息类，封装了采集装置采集到得房间信息

ComputerRoomData 继承AbstractComputerRoomData

实际房间信息类，包括温度、湿度和气压信息等。

AbstractDataGetter信息采集类，负责收集信息采集装置采集到得数据。

DataGetter继承AbstractDataGetter

在本系统中是读取装置M中的文件F。

AbstractDataTransfer信息传输类，负责将收集到得信息传输到信息中心模块。

DataTransfer继承AbstractDataTransfer

在本系统中是通过协议A以串口方式传输到信息中心。

DataCenter信息中心，负责存储采集到得所有装置的信息，并定期处理数据并存储至数据库当中

DataProcessor 信息处理器，负责处理采集到的信息，并生成统计数据

DataBaseManager 负责和数据库的连接，交互内容

ExceptionHandler 负责处理所有的异常情况。

具体的伪代码如下所示。

Class AbstractComputerRoomData{};

Class ComputerRoomData:public AbstractComputerRoomData{};

Class AbstractDataGetter{

       AbstractComputerRoomData GetData();

};

Class DataGetter:public AbstractDataGetter

{

       AbstractComputerRoomData GetData();

};

Class AbstractDataTransfer

{

       Void TransferData();

};

Class DataTransfer:public AbstractDataTransfer

{

       Void TransferData();

};

Class AbstractDataProcesser

{

       ProcessData(list<AbstractComputerData>);

};

Class DataProcesser:public AbstractDataProcesser

{

       ProcessData(list<AbstractComputerData>);

};

Class DataBaseManager

       writeRoomData(…);

};

Class MySqlDataBaseManager:public DataBaseManager

{

       writeRoomData(…);

};

Class DataCenter

{

       List<DataTransfer> m\_dataTransferlist;//保存所有的信息传输器

       DataProcesser m\_dataprocesser;//保存信息处理器

       DataBaseManager m\_dataManager;//保存数据库管理器

       ExceptionHandler m\_hander;//保存异常处理器

       List<AbstractRoomData> m\_roomDatalist;//存储获取到的房间数据

       Timer m\_timer1;//用来收集数据的定时器

       Timer m\_timer2;//用来处理数据的定时器

       Void CollectAllData();

       Void TimerProc2()

{

              Result = M\_dataprocesser.process(m\_roomDatalist);

              M\_dataManager.writeRoomData(m\_roomDatalist, result);

}

       Void TimerProc1()

       {

              SetExceptionHandler(m\_hander);

              List tmp = M\_roomDatalist = CollectAllData();

              M\_roomDatalist.add(tmp);

       }

};

2、防御式编程是提高软件质量技术的有益辅助手段。防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据。这种思想是将可能出现的错误造成的影响控制在有限的范围内。

   应注意以下方式：

**1. 在非法输入（Invalid Inputs）中保护你的程序**

**2. 断言（Assertions）**

**3. 错误处理技术**

**4. 隔栏（barricades）**

隔栏本身就是一组错误处理代码，对于内部类只需要使用断言而无需使用错误处理代码。当断言为假时，表明了问题出在了程序中而不是数据中，需要通过修改代码来消除问题。

针对内存管理的可能造成的问题，可以采用重载new和delete的方法来进行一定的防御式变成，具体方法就是自己去管理分配出来的空间，采用在分配区块的时候向系统多要一块空间，这块空间用来表示这个申请好的空间有没有被使用。用这个空间将此空间标记为未使用或已归还(可用)、已使用(不可用)的状态。

针对内存泄露的问题，可以在退出程序的时候检查还有没有申请了但没有归还的空间

针对悬挂指针的问题，可以在重载delete的函数当中检查对应的标记位，如果标记是已归还的话，说明出现了悬挂指针的问题。

**本题注意点：**以上是防御式变成所可能用到的思想和策略，这里不一定需要学生都能写出，但写出的要能说清楚。另外，上述是思想，学生答题时需要结合在题目中提出的几个可能出问题的地方，进行有针对性地答题，同时要针对这些具体问题给出清晰的解决方案，在具体评分过程中，只要写出可行的内存检查和管理方案即可，我们鼓励同学提供高效、巧妙的设计方案，但在这里只要能够给出一到两种可行方案即可。

|  |  |
| --- | --- |
| 得分 |  |

4、（本题满分　40　分）

现有一虚拟机支持以下指令集合：

DECVAR varname type   //定义变量及其类型,支持int, double

          LOADVAR varname     //将变量对应的值加载到运算堆栈中

          LOADCON constant    //将常量加载到运算堆栈中

          STOREVAR varname   //将栈顶的值出栈，并存储到符号表中

          BADD     //从运算栈顶取两个值进行相加操作，并把结果压回栈顶

          JUMP DEST //无条件跳转到目标代码

          PRINT varname //将变量的值输出到控制台

          NOP  //空指令

|  |  |
| --- | --- |
| A.op | 等价的C语言程序描述 |
| (1)     DECVAR a 1   //1表示整形  (2)     DECVAR b 2   //2表示浮点型  (3)     LOADCONS 1  (4)     STOREVAR a  (5)     LOADCONS 3.0  (6)     STOREVAR b  (7)     LOADVAR a  (8)     LOADVAR b  (9)     BADD  (10) STOREVAR a  (11) PRINT a  (12) JUMP 16  (13) LOADCONS 4.0  (14) STOREVAR b  (15) PRINT b  (16) NOP | int a;  double b;  a = 1;    b = 3.0;    a = a + b；        printf(“%i”,a);  goto a;  b=4.0;    printf(“%f”,b);  a: ; |

实现虚拟机的Virtual\_Machine类存储了需要执行的字节码，并动态地维护执行环境，如下图所示,Virtual\_Machine为当前执行的程序维护了所用变量的符号表map<string, VarValue>以及所用到的运算栈stack<VarValue>。

|  |
| --- |
| 流程图中变量-值映射（符号表）  运行时栈 |

虚拟机执行代码总体框架如以下代码所示：

int main(){

     …………………………….

     //获取需要执行的代码文件名，假设存储在file变量中

     ifstream op\_file(file); //打开代码文件，代码文件以文本方式存储，每一行为一条字节码指令

     Virtual\_Machine vm(op\_file); //构建虚拟机对象,虚拟中用vector<Instruction\*> opcodes

                             //存储指令；

     vm.loadInstructions()；//将指令加载到opcodes中，loadInstructions函数实现了对文本中指令的解析

     vm.execute();//按序依次执行opcodes中的指令。

}

说明:map类插入值对的方法,如向字符串和整数值对的map m中插入<”a”,1>：m[“a”]=1,如果该 map中没有对应的key，则插入改值对，否则覆盖改key现有的值。如果要判断map m是否存在某key,可以使用以下判断m.find(key) != m.end()。Stack类方法：top()获取栈顶元素，pop()将栈顶元素弹出，push()将元素压栈。

(1)     请按照该虚拟机的指令格式，设计Instruction对象，支持的操作数类型包括整形，浮点型。

(2)     以表驱动的方式实现Virtual\_Machine类的execute函数……并按照该接口实现 LOADVAR, STOREVAR, BADD, PRINT, NOP等指令

(3)     按照你给出的表驱动执行框架，请写出如果该虚拟机需要额外添加BMOD(取模运算指令)所需的步骤。

请注意遵循编码阶段有关变量、语句和子程序的书写方法，并以注释方式说明。

**参考答案：**

1)       设计Instruction对象

定义CVarValue

class CVarValue

{

public:

     union ActualValue

     {

         int IntVal;

         double DoubleVal;

         string StringVal;

     };

    enum EVAR\_TYPE

     {

         \_INTEGER,

         \_DOUBLE,

         \_STRING

     };

     EVAR\_TYPE m\_varType;

     ActualValue m\_value;

     CVarValue()

     {

     }

     ~CVarValue()

     {

     }

     CVarValue(int m\_value): m\_varType(\_INTEGER), m\_isValid(false), m\_memSize(-1)

     {

         this->m\_value.IntVal = m\_value;

     }

     CVarValue(double m\_value): m\_varType(\_DOUBLE), m\_isValid(false), m\_memSize(-1)

     {

         this->m\_value.DoubleVal = m\_value;

     }

     inline int GetInt() const

     {

         return m\_value.IntVal;

     }

     inline double GetDouble() const

     {

         return m\_value.DoubleVal;

     }

}

则Instruction可定义为

class Instruction

{

     int opcode;

     CVarValue para1;

     Instruction(int code): opcode(code), para2(0) {}

     Instruction(int code, const CVarValue &v): opcode(code), para1(v) {}

};

注意：一条指令主要包含操作码和操作数；我们将题目简化为只考虑单操作数或无操作数的情况。其中操作码为简单整型值，操作数的定义要有实际值和类型。不单独定义CVarValue而将CVarValue中表示实际值的union和表示类型的枚举直接写在Instruction类中也是可行的

2)       以表驱动的形式实现Virtual\_Machine类的execute函数，给出表结构的定义，描述指令处理函数的接口，并按照该接口实现题中所述五条指令的处理函数

该表结构实现了操作码（opcode）与处理函数的一个对应；可以像CTF项目中自定义一个结构，如：

typedef struct

{

    int opcode;

    HANDLER h\_f;

} OP\_MAP\_ENTRY ;

#define BEGIN\_OP\_HANDLER\_MAP(op\_handle\_map)\

    static const OP\_MAP\_ENTRY op\_handle\_map[] = \

更简单的，也可以自定义一个数组，数组下标即操作数，数组内容为处理函数的函数指针

Handler[] table = {f1, f2, …., fn};

也可使用HashMap或其他容器类实现，下面用HashMap作示例。

指令处理函数的接口为 void Handler(vm\*, Instruction&)

依次实现各指令处理函数

bool handle\_LOADVAR(CVM \*v, const Instruction &ins)

{

assert(ins.para1.m\_varType == \_STRING);

    std::string varName = std::string((char \*)(ins.para1.m\_value.StringVal));

    CVarValue \*existVarValue = v->m\_map.find(varName);

    assert(existVarValue != NULL);

    v->m\_stack.push(new CVarValue(\*existVarValue));

    return true;

}

bool handle\_STOREVAR(CVM \*v, const SInstruction &ins)

{

    assert(ins.para1.m\_varType == \_STRING);

    std::string varName = std::string((char \*)(ins.para1.m\_value.StringVal));

    CVarValue \*stackTop = v->m\_stack->top();

    v->m\_stack->pop();

    CVarValue \*existVar = v->m\_map.find(varName);

    assert(existVar != NULL);

    assert(stackTop->m\_varType == existVar->m\_varType);

    m\_map[varName] = stackTop;

    return true;

}

bool handle\_BADD(CVM \*v, const SInstruction &ins)

{

    std::cout << "ADD" << std::endl;

    CVarValue \*operand2 = v->m\_stack->top();

    v->m\_stack->pop();

    CVarValue \*operand1 = v->m\_stack->top();

    assert(operand1->m\_varType == \_INTEGER || operand1->m\_varType == \_DOUBLE);

    switch (operand1->m\_varType)

    {

    case \_INTEGER:

        operand1->m\_value.IntVal += operand2->m\_value.IntVal;

        break;

    case \_DOUBLE:

        operand1->m\_value.DoubleVal += operand2->m\_value.DoubleVal;

        break;

    default:

        ;//类型错误

    }

    delete operand2;

    return true;

}

void handle\_ PRINT (CVM \*v, const SInstruction &ins)

{

    std::string varName = std::string((char \*)(ins.para1.m\_value.StringVal));

    CVarValue \*existVar = v->m\_map.find(varName);

    assert(existVar != NULL);

    switch (operand1->m\_varType)

    {

    case \_INTEGER:

        cout<<m\_value.IntVal;

        break;

    case \_DOUBLE:

        cout<<m\_value.Double;

        break;

    default:

        ;//类型错误

    }

}

void handle\_ NOP (CVM \*v, const SInstruction &ins)

{

    ;

}

-----------------------使用hash-----------------------------------------

Instruction可定义为

class Instruction

{

       string opcode;

       CVarValue para1;

       Instruction(string code): opcode(code), para2(0) {}

       Instruction(string code, const CVarValue &v): opcode(code), para1(v) {}

};

#include<hash\_map>

bool handle\_LOADVAR(const Instruction &ins);

bool handle\_STOREVAR(const Instruction &ins)

bool handle\_BADD(const Instruction &ins)

bool handle\_PRINT(const Instruction &ins)

bool handle\_NOP(const Instruction &ins)

typedef hash\_map<string,bool (\*handle)(const Instruction &ins)> InstrMap;

InstrMap hash\_insTable;

/\*

\*the function setup\_hashtable is to setup the instuction hash table which will be referred to for use

\* in function of Virtual\_Machine::execute()

\* the function is the member function of the the class Virtual\_Machine

\*/

void setup\_hashtable()

{

       hash\_insTable["LOADVAR"] = handle\_LOADVAR;

       hash\_insTable["STOREVAR"] = handle\_STOREVAR;

       hash\_insTable["BADD"] = handle\_BADD;

       hash\_insTable["PRINT"] = handle\_PRINT;

       hash\_insTable["NOP"] = handle\_NOP;

}

void execute()

{

       vector<Instruction\*>::iterator it = opcodes.begin();

       while(it != opcodes.end())

       {

              if(hash\_insTable.find((\*it)->opcode) != hash\_insTable.end())

              {

                     (hash\_insTable[(\*it)->opcode])(\*it);

              }else

              {

                     cout<<"no identified opcode"<<(\*it)->opcode<<endl;

              }

       }

}

3)       额外添加BMOD指令时，须两步工作：

* 向表中注册BMOD处理函数；即在Handler数组中增加相应的项
* 实现BMOD的处理函数