4.1 循规蹈矩

在完成第五章后，考虑需要在之前加一章节关于 OMNeT++ 类说明，在这个仿真软件中，主要使用的语言是 C++，因此大多数数据类型是类或者结构，本章还是走其他技术书一样的老路线，注释这些数据类型，对类成员函数进行说明，可能与第五章有些重复的地方，但是其五章更多的偏向于实际应用，可能读者看过这里后，会发现 OMNeT++ 接口是真好用。

4.2 类说明

4.2.1 cModule

为了能更好的解释这个的库的使用，程序清单 4.1 为类 cModule 原型，cModule 类在 OMNeT++ 中表示一个节点的对象，这个节点可以是复合节点或者简单节点，通过这个类，程序员可以访问描述这个节点的.ned 文件中设置的参数，或者是由 omnetpp.ini 传入的参数。简而言之，我们最后就是面向这些类进行网络设计。

程序清单 4.1

```
class SIM_API cModule : public cComponent //implies noncopyable
{
    friend class cGate;
    friend class cSimulation;
    friend class cModuleType;
    friend class cChannelType;

  public:
    /*
     * 模块门的迭代器
     * Usage:
     * for (cModule::GateIterator it(module); !it.end(); ++it) {
     *     cGate *gate = *it;
     *     ...
     * }
     */
    class SIM_API GateIterator
    {
      ...
    };

    /*
```

```
     * 复合模块的子模块迭代器
     * Usage:
     * for (cModule::SubmoduleIterator it(module); !it.end(); ++it) {
     *     cModule *submodule = *it;
     *     ...
     * }
     */
    class SIM_API SubmoduleIterator
    {
      ...
    };

    /*
     * 模块信道迭代器
     * Usage:
     * for (cModule::ChannelIterator it(module); !it.end(); ++it) {
     *     cChannel *channel = *it;
     *     ...
     * }
     */
    class SIM_API ChannelIterator
    {
      ...
    };

  public:
    // internal: currently used by init
    void setRecordEvents(bool e)  {setFlag(FL_RECORD_EVENTS,e);}
    bool isRecordEvents() const  {return flags&FL_RECORD_EVENTS;}

  public:
#ifdef USE_OMNETPP4x_FINGERPRINTS
    // internal: returns OMNeT++ V4.x compatible module ID
    int getVersion4ModuleId() const { return version4ModuleId; }
#endif

    // internal: may only be called between simulations, when no modules exist
    static void clearNamePools();

    // internal utility function. Takes O(n) time as it iterates on the gates
    int gateCount() const;

    // internal utility function. Takes O(n) time as it iterates on the gates
    cGate *gateByOrdinal(int k) const;
    // internal: calls refreshDisplay() recursively
    virtual void callRefreshDisplay() override;

    // internal: return the canvas if exists, or nullptr if not (i.e. no create-on-demand)
    cCanvas *getCanvasIfExists() {return canvas;}

    // internal: return the 3D canvas if exists, or nullptr if not (i.e. no create-on-dema
    cOsgCanvas *getOsgCanvasIfExists() {return osgCanvas;}

  public:

    /** @name Redefined cObject member functions. */
    //@{

    /**
     * Calls v->visit(this) for each contained object.
     * See cObject for more details.
     */
    virtual void forEachChild(cVisitor *v) override;

    /**
     * Sets object's name. Redefined to update the stored fullName string.
     */
    virtual void setName(const char *s) override;

    /**
     * Returns the full name of the module, which is getName() plus the
     * index in square brackets (e.g. "module[4]"). Redefined to add the
     * index.
     */
    virtual const char *getFullName() const override;

    /**
     * Returns the full path name of the module. Example: <tt>"net.node[12].gen"</tt>.
     * The original getFullPath() was redefined in order to hide the global cSimulation
     * instance from the path name.
     */
    virtual std::string getFullPath() const override;

    /**
     * Overridden to add the module ID.
     */
    virtual std::string str() const override;
    //@}
```

```
/** @name Setting up the module. */
//@{

/**
 * Adds a gate or gate vector to the module. Gate vectors are created with
 * zero size. When the creation of a (non-vector) gate of type cGate::INOUT
 * is requested, actually two gate objects will be created, "gatename$i"
 * and "gatename$o". The specified gatename must not contain a "$i" or "$o"
 * suffix itself.
 *
 * CAUTION: The return value is only valid when a non-vector INPUT or OUTPUT
 * gate was requested. nullptr gets returned for INOUT gates and gate vectors.
 */
virtual cGate *addGate(const char *gatename, cGate::Type type, bool isvector=false);

/**
 * Sets gate vector size. The specified gatename must not contain
 * a "$i" or "$o" suffix: it is not possible to set different vector size
 * for the "$i" or "$o" parts of an inout gate. Changing gate vector size
 * is guaranteed NOT to change any gate IDs.
 */
virtual void setGateSize(const char *gatename, int size);

/*
 * 下面的接口是关于模块自己的信息
 */
// 复合模块还是简单模块
virtual bool isSimple() const;

/**
 * Redefined from cComponent to return KIND_MODULE.
 */
virtual ComponentKind getComponentKind() const override  {return KIND_MODULE;}

/**
 * Returns true if this module is a placeholder module, i.e.
 * represents a remote module in a parallel simulation run.
 */
virtual bool isPlaceholder() const  {return false;}

// 返回模块的父模块，对于系统模块，返回 nullptr
virtual cModule *getParentModule() const override;

/**
 * Convenience method: casts the return value of getComponentType() to cModuleType.
 */
cModuleType *getModuleType() const  {return (cModuleType *)getComponentType();}

// 返回模块属性，属性在运行时不能修改
virtual cProperties *getProperties() const override;

// 如何模块是使用向量的形式定义的，返回 true
bool isVector() const  {return vectorSize>=0;}

// 返回模块在向量中的索引
int getIndex() const  {return vectorIndex;}

// 返回这个模块向量的大小，如何该模块不是使用向量的方式定义的，返回 1
int getVectorSize() const  {return vectorSize<0 ? 1 : vectorSize;}

// 与 getVectorSize() 功能相似
_OPPDEPRECATED int size() const  {return getVectorSize();}


/*
 * 子模块相关功能
 */

// 检测该模块是否有子模块
virtual bool hasSubmodules() const {return firstSubmodule!=nullptr;}

// 寻找子模块 name，找到返回模块 ID，否则返回-1
// 如何模块采用向量形式定义，那么需要指明 index
virtual int findSubmodule(const char *name, int index=-1) const;

// 直接得到子模块 name 的指针，没有这个子模块返回 nullptr
// 如何模块采用向量形式定义，那么需要指明 index
virtual cModule *getSubmodule(const char *name, int index=-1) const;

/*
 * 一个更强大的获取模块指针的接口，通过路径获取
 *
 * Examples:
 *   "" means nullptr.
 *   "." means this module;
 *   "<root>" means the toplevel module;
 *   ".sink" means the sink submodule of this module;
 *   ".queue[2].srv" means the srv submodule of the queue[2] submodule;
 *   "^.host2" or ".^.host2" means the host2 sibling module;
 *   "src" or "<root>.src" means the src submodule of the toplevel module;
 *   "Net.src" also means the src submodule of the toplevel module, provided
 *   it is called Net.
 *
 *  @see cSimulation::getModuleByPath()
 */
virtual cModule *getModuleByPath(const char *path) const;

/*
 * 门的相关操作
 */

/**
 * Looks up a gate by its name and index. Gate names with the "$i" or "$o"
 * suffix are also accepted. Throws an error if the gate does not exist.
 * The presence of the index parameter decides whether a vector or a scalar
 * gate will be looked for.
 */
virtual cGate *gate(const char *gatename, int index=-1);

/**
 * Looks up a gate by its name and index. Gate names with the "$i" or "$o"
 * suffix are also accepted. Throws an error if the gate does not exist.
 * The presence of the index parameter decides whether a vector or a scalar
 * gate will be looked for.
 */
const cGate *gate(const char *gatename, int index=-1) const {
    return const_cast<cModule *>(this)->gate(gatename, index);
}


/**
 * Returns the "$i" or "$o" part of an inout gate, depending on the type
 * parameter. That is, gateHalf("port", cGate::OUTPUT, 3) would return
 * gate "port$o[3]". Throws an error if the gate does not exist.
 * The presence of the index parameter decides whether a vector or a scalar
 * gate will be looked for.
 */
const cGate *gateHalf(const char *gatename, cGate::Type type, int index=-1) const {
    return const_cast<cModule *>(this)->gateHalf(gatename, type, index);
}

// 检测是否有门
virtual bool hasGate(const char *gatename, int index=-1) const;

// 寻找门，如果没有返回-1，找到返回门 ID
```

```
virtual int findGate(const char *gatename, int index=-1) const;

// 通过 ID 得到门地址，目前我还没有用到过
const cGate *gate(int id) const {return const_cast<cModule *>(this)->gate(id);}

// 删除一个门（很少用）
virtual void deleteGate(const char *gatename);


//返回模块门的名字，只是基本名字 (不包括向量门的索引, "[]" or the "$i"/"$o")
virtual std::vector<const char *> getGateNames() const;

// 检测门（向量门）类型，可以标明"$i","$o"
virtual cGate::Type gateType(const char *gatename) const;

// 检测是否是向量门，可以标明"$i","$o"
virtual bool isGateVector(const char *gatename) const;

// 得到门的大小，可以指明"$i","$o"
virtual int gateSize(const char *gatename) const;

// 对于向量门，返回 gate0 的 ID 号
// 对于标量 ID，返回 ID
// 一个公式：ID = gateBaseId + index
// 如果没有该门，抛出一个错误
virtual int gateBaseId(const char *gatename) const;

/**
 * For compound modules, it checks if all gates are connected inside
 * the module (it returns <tt>true</tt> if they are OK); for simple
 * modules, it returns <tt>true</tt>. This function is called during
 * network setup.
 */
virtual bool checkInternalConnections() const;

/**
 * This method is invoked as part of a send() call in another module.
 * It is called when the message arrives at a gates in this module which
 * is not further connected, that is, the gate's getNextGate() method
 * returns nullptr. The default, cModule implementation reports an error
 * ("message arrived at a compound module"), and the cSimpleModule
 * implementation inserts the message into the FES after some processing.
 */
virtual void arrived(cMessage *msg, cGate *ongate, simtime_t t);
//@}
```

```
/*
 * 公用的
 */
// 在父模块中寻找某个参数，没找到抛出 cRuntimeError
virtual cPar& getAncestorPar(const char *parname);

/**
 * Returns the default canvas for this module, creating it if it hasn't
 * existed before.
 */
virtual cCanvas *getCanvas() const;

/**
 * Returns the default 3D (OpenSceneGraph) canvas for this module, creating
 * it if it hasn't existed before.
 */
virtual cOsgCanvas *getOsgCanvas() const;

// 设置是否在此模块的图形检查器上请求内置动画。
virtual void setBuiltinAnimationsAllowed(bool enabled) {setFlag(FL_BUILTIN_ANIMATIONS,

/**
 * Returns true if built-in animations are requested on this module's
 * graphical inspector, and false otherwise.
 */
virtual bool getBuiltinAnimationsAllowed() const {return flags & FL_BUILTIN_ANIMATIONS
//@}

/** @name Public methods for invoking initialize()/finish(), redefined from cComponent
 * initialize(), numInitStages(), and finish() are themselves also declared in
 * cComponent, and can be redefined in simple modules by the user to perform
 * initialization and finalization (result recording, etc) tasks.
 */
//@{
/**
 * Interface for calling initialize() from outside.
 */
virtual void callInitialize() override;

/**
 * Interface for calling initialize() from outside. It does a single stage
 * of initialization, and returns <tt>true</tt> if more stages are required.
 */
virtual bool callInitialize(int stage) override;
```

```
    /**
     * Interface for calling finish() from outside.
     */
    virtual void callFinish() override;


    /*
     * 动态模块创建
     */

    /**
     * Creates a starting message for modules that need it (and recursively
     * for its submodules).
     */
    virtual void scheduleStart(simtime_t t);

    // 删除自己
    virtual void deleteModule();

    // 移动该模块到另一个父模块下，一般用于移动场景。规则较复杂，可到原头文件查看使用说明
    virtual void changeParentTo(cModule *mod);
};
```

cModule 是 OMNeT++ 中用于代表一个模块的对象实体，如果你在编写网络仿真代码时，这个模块可以是简单模块或者复合模块，当需要得到这个模块相关属性时可以考虑到这个 **cModule** 类里边找找，说不定有意外的惊喜，也许有现成的函数实现你需要的功能。下面将这个类原型解剖看看：

- 迭代器：**GateIterator**

```
usage:
for (cModule::GateIterator it(module); !it.end(); ++it) {
        cGate *gate = *it;
        ...
}
```

该迭代器可用于遍历模块 **module** 的门向量，得到该门可用于其他作用。

- 迭代器：**SubmoduleIterator**

```
usage:
for (cModule::SubmoduleIterator it(module); !it.end(); ++it) {
        cModule *submodule = *it;
        ...
}
```

对于一个复合模块，包括多个简单模块或者复合模块，可使用该迭代器进行遍历操作，在第五章涉及到这个迭代器的使用。

- 迭代器：**ChannelIterator**

```
usage:
for (cModule::ChannelIterator it(module); !it.end(); ++it) {
        cChannel *channel = *it;
        ...
}
```

可用于遍历该模块的所有的信道。

4.2.2 cPar

cPar 同样是我们设置网络时不可避免的类，通过 cPar 得到节点在网络拓扑文件和配置文件中设置的参数，浏览完 cPar 所有成员函数，可以看出 cPar 基本提供了网络设计者想要的所有数据转换接口。

```
class SIM_API cPar : public cObject
{
    friend class cComponent;
  public:
    enum Type {
        BOOL = 'B',
        DOUBLE = 'D',
        LONG = 'L',
        STRING = 'S',
        XML = 'X'
    };

  private:
    cComponent *ownerComponent;
    cParImpl *p;
    cComponent *evalContext;

  private:
    // private constructor and destructor -- only cComponent is allowed to create paramete
    cPar() {ownerComponent = evalContext = nullptr; p = nullptr;}
    virtual ~cPar();
    // internal, called from cComponent
    void init(cComponent *ownercomponent, cParImpl *p);
    // internal
    void moveto(cPar& other);
    // internal: called each time before the value of this object changes.
    void beforeChange();
    // internal: called each time after the value of this object changes.
    void afterChange();

  public:
    // internal, used by cComponent::finalizeParameters()
    void read();
    // internal, used by cComponent::finalizeParameters()
    void finalize();
    // internal: applies the default value if there is one
    void acceptDefault();
    // internal
    void setImpl(cParImpl *p);
    // internal
    cParImpl *impl() const {return p;}
    // internal
    cParImpl *copyIfShared();

#ifdef SIMFRONTEND_SUPPORT
    // internal
    virtual bool hasChangedSince(int64_t lastRefreshSerial);
#endif

  public:
    /** @name Redefined cObject methods */
    //@{
    /**
     * Assignment operator.
     */
    void operator=(const cPar& other);

    // 返回参数的名字
    virtual const char *getName() const override;
```

```
// 以字符串的形式返回参数
virtual std::string str() const override;

/**
 * Returns the component (module/channel) this parameter belongs to.
 * Note: return type is cObject only for technical reasons, it can be
 * safely cast to cComponent.
 */
virtual cObject *getOwner() const override; // note: cannot return cComponent* (covari

/**
 * Calls v->visit(this) for contained objects.
 * See cObject for more details.
 */
virtual void forEachChild(cVisitor *v) override;
//@}

/** @name Type, flags. */
//@{
/**
 * Returns the parameter type
 */
Type getType() const;

/**
 * Returns the given type as a string.
 */
static const char *getTypeName(Type t);

/**
 * Returns true if the stored value is of a numeric type.
 */
bool isNumeric() const;

/**
 * Returns true if this parameter is marked in the NED file as "volatile".
 * This flag affects the operation of setExpression().
 */
bool isVolatile() const;

/**
 * Returns false if the stored value is a constant, and true if it is
 * an expression. (It is not examined whether the expression yields
 * a constant value.)
```

```
     */
    bool isExpression() const;

    /**
     * Returns true if the parameter value expression is shared among several
     * modules to save memory. This flag is purely informational, and whether
     * a parameter is shared or not does not affect operation at all.
     */
    bool isShared() const;

    /**
     * Returns true if the parameter is assigned a value, and false otherwise.
     * Parameters of an already initialized module or channel are guaranteed to
     * assigned, so this method will return true for them.
     */
    bool isSet() const;

    /**
     * Returns true if the parameter is set (see isSet()) or contains a default
     * value, and false otherwise. Parameters of an already initialized module or
     * channel are guaranteed to be assigned, so this method will return true for them.
     */
    bool containsValue() const;

    /**
     * Return the properties for this parameter. Properties cannot be changed
     * at runtime.
     */
    cProperties *getProperties() const;
    //@}

    /** @name Setter functions. Note that overloaded assignment operators also exist. */
    //@{

    /**
     * Sets the value to the given bool value.
     */
    cPar& setBoolValue(bool b);

    /**
     * Sets the value to the given long value.
     */
    cPar& setLongValue(long l);

    /**
```

```
     * Sets the value to the given double value.
     */
    cPar& setDoubleValue(double d);

    /**
     * Sets the value to the given string value.
     * The cPar will make its own copy of the string. nullptr is also accepted
     * and treated as an empty string.
     */
    cPar& setStringValue(const char *s);

    /**
     * Sets the value to the given string value.
     */
    cPar& setStringValue(const std::string& s)  {setStringValue(s.c_str()); return *this;}

    /**
     * Sets the value to the given cXMLElement.
     */
    cPar& setXMLValue(cXMLElement *node);

    /**
     * Sets the value to the given expression. This object will assume
     * the responsibility to delete the expression object.
     *
     * The evalcontext parameter determines the module or channel in the
     * context of which the expression will be evaluated. If evalcontext
     * is nullptr, the owner of this parameter will be used.
     *
     * Note: if the parameter is marked as non-volatile (isVolatile()==false),
     * one should not set an expression as value. This is not enforced
     * by cPar though.
     *
     * @see getOwner(), getEvaluationContext(), setEvaluationContext()
     */
    cPar& setExpression(cExpression *e, cComponent *evalcontext=nullptr);

    /**
     * If the parameter contains an expression (see isExpression()), this method
     * sets the evaluation context for the expression.
     *
     * @see getEvaluationContext(), isExpression(), setExpression()
     */
    void setEvaluationContext(cComponent *ctx)  {evalContext = ctx;}
    //@}
```

```
/** @name Getter functions. Note that overloaded conversion operators also exist. */
//@{

/**
 * Returns value as a boolean. The cPar type must be BOOL.
 */
bool boolValue() const;

/**
 * Returns value as long. The cPar type must be LONG or DOUBLE.
 */
long longValue() const;

/**
 * Returns value as double. The cPar type must be LONG or DOUBLE.
 */
double doubleValue() const;

/**
 * Returns the parameter's unit ("s", "mW", "Hz", "bps", etc),
 * as declared with the @unit property of the parameter in NED,
 * or nullptr if no unit was specified. Unit is only valid for LONG and DOUBLE
 * types.
 */
const char *getUnit() const;

/**
 * Returns value as const char *. The cPar type must be STRING.
 * This method may only be invoked when the parameter's value is a
 * string constant and not the result of expression evaluation, otherwise
 * an error is thrown. This practically means this method cannot be used
 * on parameters declared as "volatile string" in NED; they can only be
 * accessed using stdstringValue().
 */
const char *stringValue() const;

/**
 * Returns value as string. The cPar type must be STRING.
 */
std::string stdstringValue() const;

/**
 * Returns value as pointer to cXMLElement. The cPar type must be XML.
 *
```

```
     * The lifetime of the returned object tree is undefined, but it is
     * valid at least until the end of the current simulation event or
     * initialize() call. Modules are expected to process their XML
     * configurations at once (within one event or within initialize()),
     * and not hang on to pointers returned from this method. The reason
     * for the limited lifetime is that this method may return pointers to
     * objects stored in an internal XML document cache, and the simulation
     * kernel reserves the right to discard cached XML documents at any time
     * to free up memory, and re-load them on demand (i.e. when xmlValue() is
     * called again).
     */
    cXMLElement *xmlValue() const;

    /**
     * Returns pointer to the expression stored by the object, or nullptr.
     */
    cExpression *getExpression() const;

    /**
     * If the parameter contains an expression, this method returns the
     * module or channel in the context of which the expression will be
     * evaluated. (The context affects the resolution of parameter
     * references, and NED operators like <tt>index</tt> or <tt>sizeof()</tt>.)
     * If the parameter does not contain an expression, the return value is
     * undefined.
     *
     * @see isExpression(), setEvaluationContext()
     */
    cComponent *getEvaluationContext() const  {return evalContext;}
    //@}

    /** @name Miscellaneous utility functions. */
    //@{
    /**
     * For non-const values, replaces the stored expression with its
     * evaluation.
     */
    void convertToConst();

    /**
     * Converts the value from string, and stores the result.
     * If the text cannot be parsed, an exception is thrown, which
     * can be caught as std::runtime_error& if necessary.
     *
     * Note: this method understands expressions too, but does NOT handle
     * the special values "default" and "ask".
     */
    void parse(const char *text);
    //@}

    /** @name Overloaded assignment and conversion operators. */
    //@{

    /**
     * Equivalent to setBoolValue().
     */
    cPar& operator=(bool b)  {return setBoolValue(b);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(char c)  {return setLongValue((long)c);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(unsigned char c)  {return setLongValue((long)c);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(int i)  {return setLongValue((long)i);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(unsigned int i)  {return setLongValue((long)i);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(short i)  {return setLongValue((long)i);}

    /**
     * Converts the argument to long, and calls setLongValue().
     */
    cPar& operator=(unsigned short i)  {return setLongValue((long)i);}

    /**
     * Equivalent to setLongValue().
```

```
 */
cPar& operator=(long l)  {return setLongValue(l);}

/**
 * Converts the argument to long, and calls setLongValue().
 */
cPar& operator=(unsigned long l) {return setLongValue((long)l);}

/**
 * Equivalent to setDoubleValue().
 */
cPar& operator=(double d)  {return setDoubleValue(d);}

/**
 * Converts the argument to double, and calls setDoubleValue().
 */
cPar& operator=(long double d)  {return setDoubleValue((double)d);}

// 等同于 setStringValue() 函数
cPar& operator=(const char *s)  {return setStringValue(s);}

// 等同于 setStringValue() 函数
cPar& operator=(const std::string& s)  {return setStringValue(s);}

 // 等同于 setXMLValue() 函数
cPar& operator=(cXMLElement *node)  {return setXMLValue(node);}

operator bool() const  {return boolValue();}

operator char() const  {return (char)longValue();}

operator unsigned char() const  {return (unsigned char)longValue();}

operator int() const  {return (int)longValue();}

operator unsigned int() const  {return (unsigned int)longValue();}

operator short() const  {return (short)longValue();}

operator unsigned short() const  {return (unsigned short)longValue();}

// 返回 long 值，与 longValue() 相同
operator long() const  {return longValue();}

/**
```

```
    // 调用 longValue()，转换结果为 unsigned long 类型
     */
    operator unsigned long() const  {return longValue();}

    // 返回 double 值，与 doubleValue() 相同
    operator double() const  {return doubleValue();}

    /**
    // 调用 doubleValue()，将结果转换成 long double 类型返回
     */
    operator long double() const  {return doubleValue();}

    // 与 stringValue()
    operator const char *() const  {return stringValue();}

    // 与 stdstringValue() 功能一样
    operator std::string() const  {return stdstringValue();}

    // 与 xmlVlaue() 等同。注意：返回对象树的生命周期被限制了，具体看 xmlValue 说明。
    operator cXMLElement *() const  {return xmlValue();}
};
```

4.2.3 cGate

如果你需要在网络仿真运行时，动态实现两个节点之间的连接或者断开，那么你就需要在程序中用到这个类。

```
class SIM_API cGate : public cObject, noncopyable
{
    friend class cModule;
    friend class cModuleGates;
    friend class cPlaceholderModule;

  public:
    /**
     * Gate type
     */
    enum Type {
        NONE = 0,
        INPUT = 'I',
        OUTPUT = 'O',
        INOUT = 'B'
    };

  protected:
    // internal
    struct SIM_API Name
    {
        opp_string name;  // "foo"
        opp_string namei; // "foo$i"
        opp_string nameo; // "foo$o"
        Type type;
        Name(const char *name, Type type);
        bool operator<(const Name& other) const;
    };

  public:
    // Internal data structure, only public for technical reasons (GateIterator).
    // One instance per module and per gate vector/gate pair/gate.
    // Note: gate name and type are factored out to a global pool.
    // Note2: to reduce sizeof(Desc), "size" might be stored in input.gatev[0],
    // although it might not be worthwhile the extra complication and CPU cycles.
    //
    struct Desc
    {
        cModule *owner;
        Name *name;  // pooled (points into cModule::namePool)
        int vectorSize; // gate vector size, or -1 if scalar gate; actually allocated size
        union Gates { cGate *gate; cGate **gatev; };
        Gates input;
        Gates output;

        Desc() {owner=nullptr; vectorSize=-1; name=nullptr; input.gate=output.gate=nullptr
        bool inUse() const {return name!=nullptr;}
        Type getType() const {return name->type;}
        bool isVector() const {return vectorSize>=0;}
        const char *nameFor(Type t) const {return (t==INOUT||name->type!=INOUT) ? name->na
        int indexOf(const cGate *g) const {return (g->pos>>2)==-1 ? 0 : g->pos>>2;}
        bool deliverOnReceptionStart(const cGate *g) const {return g->pos&2;}
        Type getTypeOf(const cGate *g) const {return (g->pos&1)==0 ? INPUT : OUTPUT;}
        bool isInput(const cGate *g) const {return (g->pos&1)==0;}
        bool isOutput(const cGate *g) const {return (g->pos&1)==1;}
        int gateSize() const {return vectorSize>=0 ? vectorSize : 1;}
        void setInputGate(cGate *g) {ASSERT(getType()!=OUTPUT && !isVector()); input.gate=
        void setOutputGate(cGate *g) {ASSERT(getType()!=INPUT && !isVector()); output.gate=
        void setInputGate(cGate *g, int index) {ASSERT(getType()!=OUTPUT && isVector()); in
        void setOutputGate(cGate *g, int index) {ASSERT(getType()!=INPUT && isVector()); ou
        static int capacityFor(int size) {return size<8 ? (size+1)&~1 : size<32 ? (size+3)
    };

  protected:
    Desc *desc; // descriptor of the gate or gate vector, stored in cModule
    int pos;    // b0: input(0) or output(1); b1: deliverOnReceptionStart bit;
                // rest (pos>>2): array index, or -1 if scalar gate

    int connectionId;   // uniquely identifies the connection between *this and *nextgatep
    cChannel *channel;  // channel object (if exists)
    cGate *prevGate;    // previous and next gate in the path
    cGate *nextGate;

    static int lastConnectionId;

  protected:
    // internal: constructor is protected because only cModule is allowed to create instan
    explicit cGate();

    // also protected: only cModule is allowed to delete gates
    virtual ~cGate();

    // internal
    static void clearFullnamePool();

    // internal
    void installChannel(cChannel *chan);

    // internal
    void checkChannels() const;

#ifdef SIMFRONTEND_SUPPORT
    // internal
    virtual bool hasChangedSince(int64_t lastRefreshSerial);
#endif

  public:
    /** @name Redefined cObject member functions */
    //@{
    /*
     * 例如返回门 out
     */
    virtual const char *getName() const override;

    /*
```

```
     * 与 getName() 不同，需要返回门索引，例如 out[4]
     */
    virtual const char *getFullName() const override;

    /**
     * Calls v->visit(this) for each contained object.
     * See cObject for more details.
     */
    virtual void forEachChild(cVisitor *v) override;

    /**
     * Produces a one-line description of the object's contents.
     * See cObject for more details.
     */
    virtual std::string str() const override;

    /**
     * Returns the owner module of this gate.
     */
    virtual cObject *getOwner() const override; // note: cannot return cModule* (covariant
    //@}

    /**
     * This function is called internally by the send() functions and
     * channel classes' deliver() to deliver the message to its destination.
     * A false return value means that the message object should be deleted
     * by the caller. (This is used e.g. with parallel simulation, for
     * messages leaving the partition.)
     */
    virtual bool deliver(cMessage *msg, simtime_t at);

    /** @name Connecting the gate. */
    //@{
    /**
     * Connects the gate to another gate, using the given channel object
     * (if one is specified). This method can be used to manually create
     * connections for dynamically created modules.
     *
     * This method invokes callInitialize() on the channel object, unless the
     * compound module containing this connection is not yet initialized
     * (then it assumes that this channel will be initialized as part of the
     * compound module initialization process.) To leave the channel
     * uninitialized, specify true for the leaveUninitialized parameter.
     *
     * If the gate is already connected, an error will occur. The gate
```

```
 * argument cannot be nullptr, that is, you cannot use this function
 * to disconnect a gate; use disconnect() for that.
 *
 * Note: When you set channel parameters after channel initialization,
 * make sure the channel class is implemented so that the changes take
 * effect; i.e. the channel should either override and properly handle
 * handleParameterChange(), or should not cache any values from parameters.
 */
cChannel *connectTo(cGate *gate, cChannel *channel=nullptr, bool leaveUninitialized=fal

/**
 * Disconnects the gate, and also deletes the associated channel object
 * if one has been set. disconnect() must be invoked on the source gate
 * ("from" side) of the connection.
 *
 * The method has no effect if the gate is not connected.
 */
void disconnect();

/**
 * Disconnects the gate, then connects it again to the same gate, with the
 * given channel object (if not nullptr). The gate must be connected.
 *
 * @see connectTo()
 */
cChannel *reconnectWith(cChannel *channel, bool leaveUninitialized=false);
//@}

/** @name Information about the gate. */
//@{
/**
 * Returns the gate name without index and potential "$i"/"$o" suffix.
 */
const char *getBaseName() const;

/**
 * Returns the suffix part of the gate name ("$i", "$o" or "").
 */
const char *getNameSuffix() const;

/**
 * Returns the properties for this gate. Properties cannot be changed
 * at runtime.
 */
cProperties *getProperties() const;
```

```
/**
 * Returns the gate's type, cGate::INPUT or cGate::OUTPUT. (It never returns
 * cGate::INOUT, because a cGate object is always either the input or
 * the output half of an inout gate ("name$i" or "name$o").
 */
Type getType() const  {return desc->getTypeOf(this);}

/**
 * Returns the given type as a string.
 */
static const char *getTypeName(Type t);

/**
 * Returns a pointer to the owner module of the gate.
 */
cModule *getOwnerModule() const;

/**
 * Returns the gate ID, which uniquely identifies the gate within the
 * module. IDs are guaranteed to be contiguous within a gate vector:
 * <tt>module->gate(id+index) == module->gate(id)+index</tt>.
 *
 * Gate IDs are stable: they are guaranteed not to change during
 * simulation. (This is a new feature of \opp 4.0. In earlier releases,
 * gate IDs could change when the containing gate vector was resized.)
 *
 * Note: As of \opp 4.0, gate IDs are no longer small integers, and
 * cannot be used for iterating over the gates of a module.
 * Use cModule::GateIterator for iteration.
 */
int getId() const;

/**
 * Returns true if the gate is part of a gate vector.
 */
bool isVector() const  {return desc->isVector();}

/**
 * If the gate is part of a gate vector, returns the ID of the first
 * element in the gate vector. Otherwise, it returns the gate's ID.
 */
int getBaseId() const;

/**
```

```
     * If the gate is part of a gate vector, returns the gate's index in the vector.
     * Otherwise, it returns 0.
     */
    int getIndex() const  {return desc->indexOf(this);}

    /**
     * If the gate is part of a gate vector, returns the size of the vector.
     * For non-vector gates it returns 1.
     *
     * The gate vector size can also be obtained by calling the cModule::gateSize().
     */
    int getVectorSize() const  {return desc->gateSize();}

    /**
     * Alias for getVectorSize().
     */
    int size() const  {return getVectorSize();}

    /**
     * Returns the channel object attached to this gate, or nullptr if there is
     * no channel. This is the channel between this gate and this->getNextGate(),
     * that is, channels are stored on the "from" side of the connections.
     */
    cChannel *getChannel() const  {return channel;}

    /**
     * This method may only be invoked on input gates of simple modules.
     * Messages with nonzero length then have a nonzero
     * transmission duration (and thus, reception duration on the other
     * side of the connection). By default, the delivery of the message
     * to the module marks the end of the reception. Setting this bit will cause
     * the channel to deliver the message to the module at the start of the
     * reception. The duration that the reception will take can be extracted
     * from the message object, by its getDuration() method.
     */
    void setDeliverOnReceptionStart(bool d);

    /**
     * Returns whether messages delivered through this gate will mark the
     * start or the end of the reception process (assuming nonzero message length
     * and data rate on the channel.)
     *
     * @see setDeliverOnReceptionStart()
     */
    bool getDeliverOnReceptionStart() const  {return pos&2;}
    //@}

    /** @name Transmission state. */
    //@{
    /**
     * Typically invoked on an output gate, this method returns <i>the</i>
     * channel in the connection path that supports datarate (as determined
     * by cChannel::isTransmissionChannel(); it is guaranteed that there can be
     * at most one such channel per path). If there is no such channel,
     * an error is thrown.
     *
     * This method only checks the segment of the connection path that
     * <i>starts</i> at this gate, so, for example, it is an error to invoke
     * it on a simple module input gate.
     *
     * Note: this method searches the connection path linearly, so at
     * performance-critical places it may be better to cache its return
     * value (provided that connections are not removed or created dynamically
     * during simulation.)
     *
     * @see cChannel::isTransmissionChannel()
     */
    cChannel *getTransmissionChannel() const;

    /**
     * Like getTransmissionChannel(), but returns nullptr instead of throwing
     * an error if there is no transmission channel in the path.
     */
    cChannel *findTransmissionChannel() const;

    /**
     * Typically invoked on an input gate, this method searches the reverse
     * path (i.e. calls getPreviousGate() repeatedly) for the transmission
     * channel. It is guaranteed that there can be at most one such channel
     * per path. If no transmission channel is found, the method throws an error.
     *
     * @see getTransmissionChannel(), cChannel::isTransmissionChannel()
     */
    cChannel *getIncomingTransmissionChannel() const;

    /**
     * Like getIncomingTransmissionChannel(), but returns nullptr instead of
     * throwing an error if there is no transmission channel in the reverse
     * path.
     */
    cChannel *findIncomingTransmissionChannel() const;
    //@}

    /** @name Gate connectivity. */
    //@{

    /**
     * Returns the previous gate in the series of connections (the path) that
     * contains this gate, or nullptr if this gate is the first one in the path.
     * (E.g. for a simple module output gate, this function will return nullptr.)
     */
    cGate *getPreviousGate() const {return prevGate;}

    /**
     * Returns the next gate in the series of connections (the path) that
     * contains this gate, or nullptr if this gate is the last one in the path.
     * (E.g. for a simple module input gate, this function will return nullptr.)
     */
    cGate *getNextGate() const   {return nextGate;}

    /**
     * Returns an ID that uniquely identifies the connection between this gate
     * and the next gate in the path (see getNextGate()) during the lifetime of
     * the simulation. (Disconnecting and then reconnecting the gate results
     * in a new connection ID being assigned.) The method returns -1 if the gate
     * is unconnected.
     */
    int getConnectionId() const  {return connectionId;}

    /**
     * Return the ultimate source of the series of connections
     * (the path) that contains this gate.
     */
    cGate *getPathStartGate() const;

    /**
     * Return the ultimate destination of the series of connections
     * (the path) that contains this gate.
     */
    cGate *getPathEndGate() const;

    /**
     * Determines if a given module is in the path containing this gate.
     */
    bool pathContains(cModule *module, int gateId=-1);
```

```
/**
 * Returns true if the gate is connected outside (i.e. to one of its
 * sibling modules or to the parent module).
 *
 * This means that for an input gate, getPreviousGate() must be non-nullptr; for an ou
 * gate, getNextGate() must be non-nullptr.
 */
bool isConnectedOutside() const;

/**
 * Returns true if the gate (of a compound module) is connected inside
 * (i.e. to one of its submodules).
 *
 * This means that for an input gate, getNextGate() must be non-nullptr; for an output
 * gate, getPreviousGate() must be non-nullptr.
 */
bool isConnectedInside() const;

/**
 * Returns true if the gate fully connected. For a compound module gate
 * this means both isConnectedInside() and isConnectedOutside() are true;
 * for a simple module, only isConnectedOutside() is checked.
 */
bool isConnected() const;

/**
 * Returns true if the path (chain of connections) containing this gate
 * starts and ends at a simple module.
 */
bool isPathOK() const;
//@}

/** @name Display string. */
//@{
/**
 * Returns the display string for the gate, which controls the appearance
 * of the connection arrow starting from gate. The display string is stored
 * in the channel associated with the connection. If there is no channel,
 * this call creates an installs a cIdealChannel to hold the display string.
 */
cDisplayString& getDisplayString();

/**
 * Shortcut to <tt>getDisplayString().set(dispstr)</tt>.
```

```
     */
    void setDisplayString(const char *dispstr);
    //@}
};
```

4.2.4 cTopology

## 4.2.5 cExpression

4.2.6 EV 类

一个对调试程序有帮助的类。

```
//==========================================================================
//   CLOG.H  -  header for
//                     OMNeT++/OMNEST
//            Discrete System Simulation in C++
//
//==========================================================================

/*--------------------------------------------------------------*
  Copyright (C) 1992-2017 Andras Varga
  Copyright (C) 2006-2017 OpenSim Ltd.

  This file is distributed WITHOUT ANY WARRANTY. See the file
  `license' for details on this and other legal matters.
*--------------------------------------------------------------*/

#ifndef __OMNETPP_CLOG_H
#define __OMNETPP_CLOG_H

#include <ctime>
#include <sstream>
#include "simkerneldefs.h"

namespace omnetpp {

class cObject;
class cComponent;

/**
 * @brief Classifies log messages based on detail and importance.
 *
 * @ingroup Logging
 */
enum LogLevel
{
    /**
     * The lowest log level; it should be used for low-level implementation-specific
     * technical details that are mostly useful for the developers/maintainers of the
     * component. For example, a MAC layer protocol component could log control flow
     * in loops and if statements, entering/leaving methods and code blocks using this
     * log level.
     */
    LOGLEVEL_TRACE,

    /**
     * This log level should be used for high-level implementation-specific technical
     * details that are most likely important for the developers/maintainers of the
     * component. These messages may help to debug various issues when one is looking
     * at the code. For example, a MAC layer protocol component could log updates to
     * internal state variables, updates to complex data structures using this log level.
     */
    LOGLEVEL_DEBUG,

    /**
     * This log level should be used for low-level protocol-specific details that
     * may be useful and understandable by the users of the component. These messages
     * may help to track down various protocol-specific issues without actually looking
     * too deep into the code. For example, a MAC layer protocol component could log
     * state machine updates, acknowledge timeouts and selected back-off periods using
     * this level.
     */
    LOGLEVEL_DETAIL,

    /**
     * This log level should be used for high-level protocol specific details that
     * are most likely important for the users of the component. For example, a MAC
     * layer protocol component could log successful packet receptions and successful
     * packet transmissions using this level.
     */
    LOGLEVEL_INFO,
    /**
     * This log level should be used for exceptional (non-error) situations that
     * may be important for users and rarely occur in the component. For example,
     * a MAC layer protocol component could log detected bit errors using this level.
     */
    LOGLEVEL_WARN,

    /**
     * This log level should be used for recoverable (non-fatal) errors that allow
     * the component to continue normal operation. For example, a MAC layer protocol
     * component could log unsuccessful packet receptions and unsuccessful packet
     * transmissions using this level.
     */
    LOGLEVEL_ERROR,

    /**
     * The highest log level; it should be used for fatal (unrecoverable) errors
     * that prevent the component from further operation. It doesn't mean that
     * the simulation must stop immediately (because in such cases the code should
     * throw a cRuntimeError), but rather that the a component is unable to continue
     * normal operation. For example, a special purpose recording component may be
     * unable to continue recording due to the disk being full.
     */
    LOGLEVEL_FATAL,

    /**
     * Not a real log level, it completely disables logging when set.
     */
    LOGLEVEL_OFF,
};

/**
 * @brief For compile-time filtering of logs.
 *
 * One is free to define this macro before including <omnetpp.h>, or redefine
 * it any time. The change will affect subsequent log statements.
 * Log statements that use lower log levels than the one specified
 * by this macro will not be compiled into the executable.
 *
 * @ingroup Logging
 */
#ifndef COMPILETIME_LOGLEVEL
#ifdef NDEBUG
#define COMPILETIME_LOGLEVEL omnetpp::LOGLEVEL_DETAIL
#else
```

```
#define COMPILETIME_LOGLEVEL omnetpp::LOGLEVEL_TRACE
#endif
#endif

/**
 * @brief This predicate determines if a log statement gets compiled into the
 * executable.
 *
 * One is free to define this macro before including <omnetpp.h>, or redefine
 * it any time. The change will affect subsequent log statements.
 *
 * @ingroup Logging
 */
#ifndef COMPILETIME_LOG_PREDICATE
#define COMPILETIME_LOG_PREDICATE(object, logLevel, category) (logLevel >= COMPILETIME_LOG
#endif

/**
 * @brief This class groups logging related functionality.
 *
 * @see LogLevel
 * @ingroup Logging
 */
class SIM_API cLog
{
  public:
    typedef bool (*NoncomponentLogPredicate)(const void *object, LogLevel logLevel, const
    typedef bool (*ComponentLogPredicate)(const cComponent *object, LogLevel logLevel, cons

  public:
    /**
     * This log level specifies a globally applied runtime modifiable filter. This is
     * the fastest runtime filter, it works with a simple integer comparison at the call
     * site.
     */
    static LogLevel logLevel;

    /**
     * This predicate determines if a log statement is executed for log statements
     * that occur outside module or channel member functions. This is a customization
     * point for logging.
     */
    static NoncomponentLogPredicate noncomponentLogPredicate;

    /**
```

```
     * This predicate determines if a log statement is executed for log statements
     * that occur in module or channel member functions. This is a customization
     * point for logging.
     */
    static ComponentLogPredicate componentLogPredicate;

  public:
    /**
     * Returns a human-readable string representing the provided log level.
     */
    static const char *getLogLevelName(LogLevel logLevel);

    /**
     * Returns the associated log level for the provided human-readable string.
     */
    static LogLevel resolveLogLevel(const char *name);

    static inline bool runtimeLogPredicate(const void *object, LogLevel logLevel, const cha
    { return noncomponentLogPredicate(object, logLevel, category); }

    static inline bool runtimeLogPredicate(const cComponent *object, LogLevel logLevel, con
    { return componentLogPredicate(object, logLevel, category); }

    static bool defaultNoncomponentLogPredicate(const void *object, LogLevel logLevel, cons
    static bool defaultComponentLogPredicate(const cComponent *object, LogLevel logLevel, c
};

// Creates a log proxy object that captures the provided context.
// This macro is internal to the logging infrastructure.
//
// NOTE: the (void)0 trick prevents GCC producing statement has no effect warnings
// for compile time disabled log statements.
//
#define OPP_LOGPROXY(object, logLevel, category) \
    ((void)0, !(COMPILETIME_LOG_PREDICATE(object, logLevel, category) && \
    omnetpp::cLog::runtimeLogPredicate(object, logLevel, category))) ? \
    omnetpp::cLogProxy::dummyStream : omnetpp::cLogProxy(object, logLevel, category, __FIL


// Returns nullptr. Helper function for the logging macros.
inline void *getThisPtr() {return nullptr;}

/**
 * @brief Use this macro when logging from static member functions.
 *
 * Background: EV_LOG and derived macros (EV_INFO, EV_DETAIL, etc) will fail
 * to compile when placed into static member functions of cObject-derived classes
 * ("cannot call member function 'cObject::getThisPtr()' without object" in GNU C++,
 * and "C2352: illegal call of non-static member function" in Visual C++).
 * To fix it, add this macro at the top of the function; it contains local declarations
 * to make the code compile.
 *
 * @ingroup Logging
 * @hideinitializer
 */
#define EV_STATICCONTEXT  void *(*getThisPtr)() = omnetpp::getThisPtr;

/**
 * @brief This is the macro underlying EV_INFO, EV_DETAIL, EV_INFO_C, and
 * similar log macros.
 *
 * This macro should not be used directly, but via the logging macros
 * EV, EV_FATAL, EV_ERROR, EV_WARN, EV_INFO, EV_DETAIL, EV_DEBUG, EV_TRACE,
 * and their "category" versions EV_C, EV_FATAL_C, EV_ERROR_C, EV_WARN_C,
 * EV_INFO_C, EV_DETAIL_C, EV_DEBUG_C, EV_TRACE_C.
 *
 * Those macros act as C++ streams: one can write on them using the
 * left-shift (<<) operator. Their names refer to the log level they
 * represent (see LogLevel). The "category" (_C) versions accept a category
 * string. Each category acts like a separate log channel; for example,
 * one can use the "test" category to log text intended for consumption
 * by an automated test suite.
 *
 * Log statements are wrapped with compile-time and runtime guards at the
 * call site to efficiently prevent unnecessary computation of parameters
 * and log content. Compile-time guards are COMPILETIME_LOGLEVEL and
 * COMPILETIME_LOG_PREDICATE. Runtime guards (runtime log level) can be
 * set up via omnetpp.ini.
 *
 * Under certain circumstances, compiling log statements may result in errors.
 * When that happens, it is possible that the EV_STATICCONTEXT macro needs to
 * be added to the code; please review its documentation for more info.
 *
 * Examples:
 *
 * \code
 * EV_INFO << "Connection setup complete" << endl;
 * EV_INFO_C("test") << "ESTAB" << endl;
 * \endcode
 *
 * @see LogLevel, EV_STATICCONTEXT, EV_INFO, EV_INFO_C, COMPILETIME_LOGLEVEL, COMPILETIME_
 * @ingroup Logging
 * @hideinitializer
 */
#define EV_LOG(logLevel, category) OPP_LOGPROXY(getThisPtr(), logLevel, category).getStrea

/**
 * @brief Short for EV_INFO.
 * @see EV_INFO @ingroup Logging
 */
#define EV        EV_INFO

/**
 * @brief Pseudo-stream for logging local fatal errors. See EV_LOG for details.
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_FATAL  EV_LOG(omnetpp::LOGLEVEL_FATAL, nullptr)

/**
 * @brief Pseudo-stream for logging local recoverable errors. See EV_LOG for details.
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_ERROR  EV_LOG(omnetpp::LOGLEVEL_ERROR, nullptr)

/**
 * @brief Pseudo-stream for logging warnings. See EV_LOG for details.
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_WARN   EV_LOG(omnetpp::LOGLEVEL_WARN, nullptr)

/**
 * @brief Pseudo-stream for logging information with the default log level. See EV_LOG for
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_INFO   EV_LOG(omnetpp::LOGLEVEL_INFO, nullptr)

/**
 * @brief Pseudo-stream for logging low-level protocol-specific details. See EV_LOG for de
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_DETAIL EV_LOG(omnetpp::LOGLEVEL_DETAIL, nullptr)

/**
 * @brief Pseudo-stream for logging state variables and other low-level information. See E
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_DEBUG  EV_LOG(omnetpp::LOGLEVEL_DEBUG, nullptr)

/**
 * @brief Pseudo-stream for logging control flow information (entering/exiting functions,
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_TRACE  EV_LOG(omnetpp::LOGLEVEL_TRACE, nullptr)

/**
 * @brief Short for EV_INFO_C.
 * @see EV_INFO_C @ingroup Logging
 */
#define EV_C(category)        EV_INFO_C(category)

/**
 * @brief Pseudo-stream for logging local fatal errors of a specific category. See EV_LOG
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_FATAL_C(category)  EV_LOG(omnetpp::LOGLEVEL_FATAL, category)

/**
 * @brief Pseudo-stream for logging local recoverable errors of a specific category. See E
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_ERROR_C(category)  EV_LOG(omnetpp::LOGLEVEL_ERROR, category)

/**
 * @brief Pseudo-stream for logging warnings of a specific category. See EV_LOG for detail
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_WARN_C(category)   EV_LOG(omnetpp::LOGLEVEL_WARN, category)

/**
 * @brief Pseudo-stream for logging information with the default log level of a specific c
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_INFO_C(category)   EV_LOG(omnetpp::LOGLEVEL_INFO, category)

/**
 * @brief Pseudo-stream for logging low-level protocol-specific details of a specific cate
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_DETAIL_C(category) EV_LOG(omnetpp::LOGLEVEL_DETAIL, category)
```

```
/**
 * @brief Pseudo-stream for logging state variables and other low-level information of a s
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_DEBUG_C(category)  EV_LOG(omnetpp::LOGLEVEL_DEBUG, category)

/**
 * @brief Pseudo-stream for logging control flow information (entering/exiting functions,
 * @see EV_LOG @hideinitializer @ingroup Logging
 */
#define EV_TRACE_C(category)  EV_LOG(omnetpp::LOGLEVEL_TRACE, category)

/**
 * @brief This class holds various data that is captured when a particular
 * log statement executes. It also contains the text written to the log stream.
 *
 * @see cEnvir::log(cLogEntry*)
 * @ingroup Internals
 */
class SIM_API cLogEntry
{
  public:
    // log statement related
    LogLevel logLevel;
    const char *category;

    // C++ source related (where the log statement appears)
    const void *sourcePointer;
    const cObject *sourceObject;
    const cComponent *sourceComponent;
    const char *sourceFile;
    int sourceLine;
    const char *sourceFunction;

    // operating system related
    clock_t userTime;

    // the actual text of the log statement
    const char *text;
    int textLength;
};


//
// This class captures the context where the log statement appears.
// NOTE: This class is internal to the logging infrastructure.
//
class SIM_API cLogProxy
{
  private:
    // This class is used for buffering the text content to be able to send whole
    // lines one by one to the active environment.
    class LogBuffer : public std::basic_stringbuf<char> {
      public:
        LogBuffer() { }
        bool isEmpty() { return pptr() == pbase(); }
      protected:
        virtual int sync() override;  // invokes getEnvir()->log() for each log line
    };

    // act likes /dev/null
    class nullstream : public std::ostream {
      public:
        nullstream() : std::ostream(nullptr) {}  // results in rdbuf==0 and badbit==true
    };

  public:
    static nullstream dummyStream; // EV evaluates to this when in express mode (getEnvir(

  private:
    static LogBuffer buffer;  // underlying buffer that contains the text that has been wr
    static std::ostream stream;  // this singleton is used to avoid allocating a new strea
    static cLogEntry currentEntry; // context of the current (last) log statement that has
    static LogLevel previousLogLevel; // log level of the previous log statement
    static const char *previousCategory; // category of the previous log statement

  private:
    void fillEntry(LogLevel logLevel, const char *category, const char *sourceFile, int sou

  public:
    cLogProxy(const void *sourcePointer, LogLevel logLevel, const char *category, const cha
    cLogProxy(const cObject *sourceObject, LogLevel logLevel, const char *category, const c
    cLogProxy(const cComponent *sourceComponent, LogLevel logLevel, const char *category, c
    ~cLogProxy();

    std::ostream& getStream() { return stream; }
    static void flushLastLine();
};

}  // namespace omnetpp
```

```
#endif
```

4.3 虚函数

4.3.1 initialize 函数

4.3.1 handleMessage 函数

4.3.1 refreshDisplay 函数

4.3.1 finish 函数