Неявные интерфейсы C++ шаблонов.
Сложно о простом.

Полянин М.А.
28 июня 2016.

1. Сначала о дизайне.

На нынешнем уровне понимания программирования трудно написать хороший код, но можно делать объяснимый код, устраняющий известные проблемы.

Рассмотрим частный пример. Сначала это была вполне безобидная программа.

```
        switch ( hookStruct->vkCode )
        {
            case KeyWheelUp1:
            case KeyWheelUp2:
            {
                if (wParam == WM_SYSKEYDOWN || wParam == WM_KEYDOWN )
                {
                    mouse_event( MOUSEEVENTF_WHEEL, 0, 0, WHEEL_DELTA, 0 );
                }
            }
            break;

            case KeyWheelDown1:
            case KeyWheelDown2:
            {
```

```
                    if (wParam == WM_SYSKEYDOWN || wParam == WM_KEYDOWN )
                    {
                         mouse_event( MOUSEEVENTF_WHEEL, 0, 0, -WHEEL_DELTA, 0 );
                    }
               }
               break;
          }
```

Затем мы улучшили эту программу, добавив немного больше функциональности и где то внутри новой программы получили вот такой участок кода, который в принципе делает то же самое, что и предыдущий пример.

```
//mouse wheel
#ifdef Nmj_Va_mw
class Va_mw: public Vaction{
public:
     void proceed();

     Va_mw():key_timer_passed(0){}

protected:
     //what mouse event to do in child
     virtual
     void      m_event()=0;

     //repeat time counter
     unsigned key_timer_passed;
};

//mouse wheel up
```

```
class Te_mw_up: public Va_mw{
public:
     //on hold
     void m_event();
};

//mouse wheel dn
class Te_mw_dn: public Va_mw{
public:
     //on hold
     void m_event();
};
#endif
```

Этот пример страдает проблемой дизайна C++ классов, которую в общем и кратко можно описать как
"наследование используется не для реализации интерфейса".
Такая проблема затрудняет
- повторное использование кода;
- создание вариаций кода;
и приводит к комбинаторному росту числа классов.

Это серьезный изъян в дизайне, привычка применять который должна устраняться всегда, несмотря на то,
что "так сделать легче", так как иначе видеть эту проблему в более сложной программе будет еще труднее.

Почему в принципе такая проблема происходит? Достаточно попытаться создать любую объектную программу,
чтобы обнаружить, что в программе всегда есть две системы классов:
- одна система классов описывает прикладную задачу в терминах и сущностях прикладной задачи;
- вторая система классов это классы разработанные для повторного использования и описывают проблему повторного
использования в терминах и сущностях повторного использования.
Это абсолютно несовместимые задачи и интерфейсы.

Поэтому в общем в любой объектной программе всегда возникает система классов прикладной задачи,
которая через адаптеры реализуется на базе системы классов для повторного использования.

2. Почему происходит изменение дизайна.

Происходит изменение дизайна потому, что мы заранее не знаем как будут реализовываться методы класса,
во время реализации разных классов "событие мыши", обнаружив что у нас есть "логически устойчиво общий" шаблон кода,
мы его немедленно выделяем в отдельную структурную единицу C++, чтобы избавиться от дублирования кода
(особенно кода в процессе разработки).

В данном примере два разных класса реализующие общий интерфейс, были слиты в один класс "Va_mw", содержащий этот общий
шаблон кода, и этот класс был еще раз пронаследован для реализации в потомках вариации кода этого шаблона кода.
Вот это наследование и неправильно с точки зрения системы классов для повторного использования.

Для того чтобы исправить проблему дизайна, надо создать новый адаптер с интерфейсом для реализации этих вариаций
и реализовать этот адаптер нашим кодом, т.е. надо создать новый независимый интерфейс и наследовать уже его.

Второй вариант системы классов исправляет этот недостаток.

```
#ifdef Nmj_Va_mw2
class Va_mw_event;

class Va_mw: public Vaction{
protected:
     typedef
          Va_mw_event
          Vevent;
};

class Ta_mw: public Va_mw{
public:
     void proceed();

     Ta_mw(Vevent *e):
          key_timer_passed(0),
```

```
        do_event(e)
        {}

protected:
    //what mouse event to do
    Vevent        *do_event;

    //repeat time counter
    unsigned key_timer_passed;
};

//mouse wheel event
class Va_mw_event{
public:
    //what mouse event to do
    virtual
    void m_event()=0;

public:
    virtual
    ~Va_mw_event(){}
};

//mouse wheel up
class Te_mw_up: public Va_mw_event{
public:
    //on hold
    void m_event();
};
```

```
//mouse wheel dn
class Te_mw_dn: public Va_mw_event{
public:
     //on hold
     void m_event();
};
#endif
```

Появился новый интерфейс "Va_mw_event", который реализуется наследованием и класс "Va_mw" готов использовать любой потомок "Va_mw_event". Число классов (не считая класса для описания интерфейса "Va_mw_event") не изменилось, но возможности повторного использования существенно поменялись, теперь можно изменять реализацию "Va_mw_event" независимо от "Va_mw".

Осталась последняя проблема - как независимо от "Va_mw_event" уметь менять реализацию "Va_mw"? Для этого реализацию "Va_mw" надо наследованием перенести в потомки "Va_mw". Интерфейс не должен содержать код способный к вариациям.

После всего этого число классов (не считая класс для описания интерфейса "Va_mw") опять не изменилось, но как изменились возможности повторного использования для новой структуры классов!

3. Третий момент.

Теперь надо отметить что "Va_mw" особо ничего не содержит:
- ни определения класса "Va_mw_event"
- ни ссылку на "Va_mw_event" такого вида

```
protected:
     //what mouse event to do
     virtual
     Va_mw_event   *do_event() =0;
```

Это потому, что использование "Va_mw_event" зависит от потомка "Va_mw". Наличие множества простых и разрозненных классов это естественная структура классов для повторного использования, она порождает трудности в сборе из таких частей готовых

классов и необходимость создавать классы, которые занимаются деятельностью обеспечения совместной работы таких простых классов.

В данном случае чтобы указать на связь "Va_mw_event" и "Va_mw" мы используем указание на типы

```
protected:
     typedef
          Va_mw_event
          Vevent;
```

4. Четвертый момент.

Оба варианта используют виртуальные функции m_event() для реализации интерфейса, в то время как нам не требуется использовать эти классы для обработки их в общем шаблоне кода, т.е. мы можем воспользоваться шаблоном времени компиляции задаваемым с помощью директив template.

```
#ifdef Nmj_Va_mw3
//mouse wheel
template<class Tevent>
class Ta_mw: public Vaction{
public:
     void proceed();

     Ta_mw():key_timer_passed(0){}

protected:
     //what mouse event to do
     Tevent        do_event;

     //repeat time counter
     unsigned key_timer_passed;
```

```
};
#endif
```

В качестве "Tevent" можно использовать предыдущие классы "Te_mw_up" и "Te_mw_dn", заданные как "class Te_mw_up: public Va_mw_event", в этом случае метод "Ta_mw::proceed" будет все равно невиртуально вызывать метод "Te_mw_up::m_event()".

Или можно создать новые классы "Tevent_mw_up" и "Tevent_mw_dn", имеющие метод с именем "m_event".

```
#ifdef Nmj_Va_mw4
//mouse wheel up
class Tevent_mw_up{
public:
     //on hold
     inline
     void m_event();
};

//mouse wheel dn
class Tevent_mw_dn{
public:
     //on hold
     inline
     void m_event();
};
#endif
```

Создавать такие новые классы имеет смысл если известна реализация метода "m_event" и эта реализация не склонна к вариациям и является inline, что на момент создания интерфейса обычно не известно и написание таких классов обычно может трактоваться как "избыточная (premature) оптимизация".

Реализации inline методов должны быть при каждой компиляции помещены в заголовке "hpp", содержащем код, например они могут быть помещены в том же заголовке, где находится метод "Ta_mw::proceed", если характер работы и набор данных для "proceed" и "m_event" хоть сколько нибудь схож.

5. template и дублирование кода и данных.

Неправильное использование template приводит к сильному дублированию кода и данных. Шаблоны времени компиляции используются правильно, если применены для:
- описания типов пользователя C, если эти типы не используются в шаблонах времени выполнения;
- описания классов C++, если эти классы не содержат кода и не используются в шаблонах времени выполнения;
- замены директив препроцессора для условной компиляции, т.е. когда во время компиляции программы шаблон инстанцируется только один раз;
- генерации кода, который не используется повторно, независимо от linkage этого кода (inline или call).

В нашем случае мы знаем, что метод "Ta_mw::proceed()" содержит два условных выражения и может быть использован как "код который не предназначен для повторного использования". Но в общем случае предыдущий пример порождает избыточное дублирование кода "Ta_mw::proceed()".

Когда мы устранили неправильное применение наследования, введя два интерфейса "Va_mw" и "Va_mw_event" и реализуя наследованием уже их, мы избавились от "невозможности с помощью наследования описать комбинаторное сочетаний свойств".

Но в принципе из полученных нами блоков их соединением можно создать комбинаторное число сочетаний. Вот для описания таких сочетаний и можно использовать шаблоны времени компиляции, потому что все эти сочетания можно описать одним единственным таким шаблоном, который хоть и не является столь декларативным как описание интерфейса (что за класс мы вообще не считаем), но является простым адаптером этих "блоков" для какого то любого интерфейса, например для "Vaction".

```
#ifdef Nmj_Va_mw5
template<class Ta_mw, class Te_mw>
class Ta_mwe: public Vaction{
public:
    #if 0
    void proceed(){ a_mw.Ta_mw::proceed(&e_mw); }
```

```
    #else
    void proceed(){ a_mw.Ta_mw::proceed(); }

    Ta_mwe():a_mw(&e_mw){}
    Ta_mwe(const Ta_mwe& obj):a_mw(obj.a_mw), e_mw(obj.e_mw){ a_mw.do_event= &e_mw; }
    Ta_mwe& operator=(const Ta_mwe& obj){ if(&obj != this){ a_mw=obj.a_mw; e_mw=obj.e_mw;
a_mw.do_event= &e_mw; } return *this; }
    #endif

protected:
    Te_mw    e_mw;
    Ta_mw    a_mw;
};
#endif
```

Сразу же отметим, что использование постоянных взаимных ссылок между несколькими объектами разных классов, работающих вместе во время выполнения, создает сложности с поддержанием целостности ссылок при согласованной инициализации и совместном копировании этих объектов, а также с владением этими объектами.

Одним из решений проблемы является то, что при каждом вызове методов одного объекта ему передаются ссылки на все те объекты, с которыми он связан.

Это не только дает лишние параметры при вызове методов, но и говорит что класс "Ta_mw", нуждающийся в ссылке на "Va_mw_event", не может следовать интерфейсу "Vaction", в частности
- не может следовать новому методу "Vaction::proceed", который должен быть вызван с параметром "Va_mw_event*";
- или должен расширять "Vaction" доступом к полю "Va_mw_event *do_event",
  но при этом расширении все равно не может без принципиальных проблем следовать заданному для "Vaction" типу "copyable" (тип конструкторов и копирования).

И пример этого шаблона показывает, что для создания комбинаторного числа классов таким шаблоном, класс "Ta_mw", нуждающийся в ссылке на "Va_mw_event", не может следовать интерфейсу "Vaction".

Для использования "Ta_mw" в качестве "Vaction" нужен отдельный адаптер и его роль также может выполнить

этот шаблон времени компиляции, который изначально служит для описания комбинаторного сочетания частей
класса "Vaction".

Пока расширим "Vaction" доступом к полю "Va_mw_event *do_event" для "Ta_mw":

```
//protected:
public:
    //what mouse event to do
    Vevent        *do_event;
```

6. Не типизированные параметры шаблонов в C++.

В случае применения template, параметры шаблона в C++ (в нынешней версии C++) не могут быть типизированы,
т.е. для них не может быть задан интерфейс и все ошибки, даже такие грубые как отсутствие в подставленном классе
нужных имен, не говоря уже о логическом соответствии этих имен предполагаемому интерфейсу параметра,
могут быть обнаружены только во время инстанцирования кода подстановки конкретного класса в шаблоне,
а не во время декларации использования такого конкретного класса как параметра этого шаблона.

Эти этапы инстанцирования кода и декларации использования класса могут быть значительно разделаны
во времени и в пространстве.

Такое разделение затрудняет, например, написание отлаженных классов, поскольку их невозможно проверить
до применения шаблона в каждом конкретном случае, что приводит к тому, что отлаженные классы, оформленные
через template, нуждаются в исправлениях уже в момент их использования в качестве надежной библиотеки.

Нельзя сказать что интерфейс параметра шаблона в C++ задать совсем нельзя. Этот интерфейс задается неявно,
что полностью аналогично высказыванию о модулях в языке C:
"модули в языке C сделать можно, но они не поддержаны языком C".

Для явного задания интерфейса параметров шаблона в C++, можно использовать следующий предлагаемый способ описания
такого интерфейса.

```
#ifdef Nmj_Va_mw6
//Ta_mwe::Tevent compile time interface
template<class Tevent>
```

```
class Ia_mwe_event{
public:
     //void   m_event();
     typedef
          void
          (Tevent::* f_m_event)
          ();

public:
     Ia_mwe_event(){
          f_m_event m_event= &Tevent::m_event;
     }
};

//Ta_mwe::Taction compile time interface
template<class Taction>
class Ia_mwe_action{
public:
     //void   proceed();
     typedef
          void
          (Taction::* f_proceed)
          ();

     //Va_mw_event *do_event;
     typedef
          Va_mw_event
          Taction::* Tdo_event;
```

```
public:
     Ia_mwe_action(){
          f_proceed proceed= &Taction::proceed;
          Tdo_event do_event= &Taction::do_event;
     }
};

//mouse wheel
template<class Taction, class Tevent>
class Ta_mwe: public Vaction{
public:
     void proceed(){ a_mw.Taction::proceed(); }

     Ta_mwe():a_mw(&e_mw){}
     Ta_mwe(const Ta_mwe& obj):a_mw(obj.a_mw), e_mw(obj.e_mw){ a_mw.do_event= &e_mw; }
     Ta_mwe& operator=(const Ta_mwe& obj){ if(&obj!=this){ a_mw=obj.a_mw; e_mw=obj.e_mw;
a_mw.do_event= &e_mw; } return *this; }

protected:
     Tevent         e_mw;
     Taction        a_mw;

protected:
     Ia_mwe_action<Taction>
                    anIaction;

     Ia_mwe_event<Tevent>
                    anIevent;
};
```

```
#endif
```

Классы "Ia_mwe_action" и "Ia_mwe_event" описывают интерфейс параметров "Taction" и "Tevent" соответственно.

Для того чтобы включить проверку интерфейса параметра для любого класса, надо вставить объект типа интерфейса
как член класса, в нашем случае это "anIaction" и "anIevent".

Если ваш компилятор работает рационально, то он не станет транслировать никакой код, который указан в телах функции
шаблона до момента инстанцирования шаблона и использования именно этой функции, потому что в общем этот код должен
быть размещен в отдельном "hpp" файле и указан он прямо в классе просто по каким-то файловым причинам.

Поэтому вызвать код можно автоматически, если поместить этот код в конструктор. Переменные, которые этот код
инициализирует, не используются и при оптимизации этот код инициализации будет исключен, но до этого будет проверен
и прокомпилирован. Переменные помогают проверить тип возвращаемого адреса.

Поскольку задание интерфейса параметров шаблона не поддержано языком C++, то это низкоуровневый и не самый
оптимальный вариант проверки параметров шаблона, но этот вариант дает такую проверку параметров шаблонов
во время создания экземпляров шаблона класса, а не вовремя генерации кода методов шаблона класса,
что практически бывает почти так же хорошо, как проверка интерфейса параметра шаблона во время декларации класса.

Включить проверку интерфейса параметров шаблона необходимо как минимум в классе "Ta_mwe",
но нормально надо включить такую проверку "интерфейса времени компиляции класса" во всех тех классах,
которые будут использоваться в качестве параметров шаблонов с таким интерфейсом,
такое включение означает явное указание что класс имеет такой "интерфейс времени компиляции класса".

Точно также, для того чтобы включить проверку "интерфейса времени компиляции класса" для любого класса,
надо вставить объект типа интерфейса как член такого класса. Ниже пример таких вставок для классов
"Tevent_mw_up" и "Tevent_mw_dn"

```
#ifdef Nmj_Va_mw7
//mouse wheel up
class Tevent_mw_up{
public:
    //on hold
```

```
    void m_event();

protected:
    Ia_mwe_event<Tevent_mw_up>
                anIevent;
};

//mouse wheel dn
class Tevent_mw_dn{
public:
    //on hold
    void m_event();

protected:
    Ia_mwe_event<Tevent_mw_dn>
                anIevent;
};
#endif
```

Этот способ задания интерфейса времени компиляции не позволяет только по именам и типам контролировать сложное логическое поведение методов пользователя или конструкторов копирования и инициализации класса.

Сложное поведение можно задавать описывая в классе специальные имена типов или специализируя именем класса общепринятые имена шаблонов (которые описывают популярное поведение), определяя внутри специализации имена типов как разрешающие признаки для имени класса.

7. Соберем все части в одну.

Ну и теперь все эти части (проверка типов языка пользователем) в одном целом.

Признаем, что "Ta_mw" не может следовать "Vaction" по особенностям копирования и по использованию "Va_mw_event*" и для подстановки объекта "Ta_mw" в код для "Vaction" необходим адаптер для класса "Ta_mw" в класс "Vaction".

Чтобы выделить особенности копирования "Ta_mw" определим в этом классе тип "Tcopy_as_Va_mw", который своим именем указывает на правильные применения этих особенностей в шаблонах предназначенных для "Ta_mw".

Также переименуем "Va_mw_event" в "Ve_mw" и добавим немного непринципиальных модификаций.

А еще выразим код метода "Ta_mw::proceed" как шаблон, не зависящий от "Ve_mw". Надо признать что такой шаблон заставил нас виртуализировать оператор ".do_event" с помощью функций доступа, внося модификации в сложившийся интерфейс, и такое будет с каждым, кто оставит в интерфейсе указатель, не будучи достаточно уверенным, что это хранилище никогда не станет изменяться.

```
#define Nmj_Va_mw2
#define Nmj_Va_mw4
#define Nmj_Va_mw6
#define Nmj_Va_mw7
#define Nmj_Va_mw9

//var 2
//mouse wheel
#ifdef Nmj_Va_mw2
class Ve_mw;

//Ta_mw run time interface
class Va_mw{
public:
    virtual
    void     proceed() =0;

public:
    //get-set Vevent* to support Vevent* adapter
    //Vevent* pointer is non in heap
```

```
    typedef
        Ve_mw
        Vevent;

    virtual
    Vevent*       get_event()const =0;
    virtual
    void     set_event(Vevent *const) =0;

public:
    virtual
    ~Va_mw(){}

//protected: - public for Ia_mw<>
public:
    //copy logic
    typedef bool Tcopy_as_Va_mw;
};

#ifdef Nmj_Va_mw7
//Ta_mw compile time interface
template<class Taction_mv>
class Ia_mw{
public:
    //typedef bool Tcopy_as_Va_mw;
    typedef
        typename Taction_mv::Tcopy_as_Va_mw
        Tcopy_as_Va_mw;
```

```
    //void   proceed();
    typedef
        void
        (Taction_mv::* f_proceed)
        ();

    //Vevent *do_event;
    typedef
        typename Taction_mv::Vevent
        Vevent;

    typedef
        Vevent*
        (Taction_mv::* f_get_event)
        ()const;

    typedef
        void
        (Taction_mv::* f_set_event)
        (Vevent *const);

public:
    Ia_mw(){
        f_proceed proceed= &Taction_mv::proceed;
        f_get_event get_event= &Taction_mv::get_event;
        f_set_event set_event= &Taction_mv::set_event;
    }
};
#endif
```

```
#ifdef Nmj_Va_mw7
//Tmw_p compile time interface
template<class Tproceed>
class Imw_p{
public:
     //typedef bool Tcopy_as_Tmw_p;
     typedef
          typename Tproceed::Tcopy_as_Tmw_p
          Tcopy_as_Tmw_p;

     //void   proceed();
     typedef
          void
          (Tproceed::* f_proceed)
          ();

     //Vevent *do_event;
     typedef
          typename Tproceed::Vevent*
          Tproceed::* Tdo_event;

public:
     Imw_p(){
          f_proceed proceed= &Tproceed::proceed;
          Tdo_event do_event= &Tproceed::do_event;
     }
};
#endif
```

```
//Va_mv::proceed template
#ifdef Nmj_Va_mw9
template<class Tevent>
class Tmw_p{
public:
     void proceed();

public:
     //copy logic
     typedef bool Tcopy_as_Tmw_p;

     Tmw_p(Tevent *e=0):
          do_event(e),
          key_timer_passed(0)
          {}

     Tmw_p(const Tmw_p& obj, Tevent *e=0):
          do_event(e),
          key_timer_passed(obj.key_timer_passed)
          {}

     Tmw_p& operator= (const Tmw_p& obj){
          if( &obj != this){ do_event=0; key_timer_passed= obj.key_timer_passed; }
          return *this;
          }

public:
     //what mouse event to do
```

```
    Tevent        *do_event;
    typedef
        Tevent
        Vevent;

protected:
    //repeat time counter
    unsigned key_timer_passed;

protected:
    #ifdef Nmj_Va_mw7
    Imw_p<Tmw_p>
                  anImw_p;
    #endif
};
#endif

//
class Ta_mw: public Va_mw{
public:
    void     proceed();
    Vevent*       get_event()const;
    void     set_event(Vevent *const);

public:
    //copy logic
    Ta_mw(Vevent *e=0):mw_p(e){}
    Ta_mw(const Ta_mw& obj, Vevent *e=0):mw_p(obj.mw_p,e){}
    Ta_mw& operator= (const Ta_mw& obj){ if( &obj != this){ mw_p=obj.mw_p; } return *this; }
```

```
public:
     //proceed templated implementation
     typedef
          Va_mw::Vevent
          Vevent;
     typedef
          Tmw_p<Vevent>
          Tproceed;

     Tproceed
                    mw_p;

protected:
     #ifdef Nmj_Va_mw7
     Ia_mw<Ta_mw>
                    anIa_mw;
     Imw_p<Tproceed>
                    anImw_p;
     #endif
};

//mouse wheel event
#ifdef Nmj_Va_mw7
//Ve_mw compile time interface
template<class Tevent>
class Ie_mw{
public:
     //void   m_event();
```

```
     typedef
          void
          (Tevent::* f_m_event)
          ();

public:
     Ie_mw(){
          f_m_event m_event= &Tevent::m_event;
     }
};
#endif

//Te_mw run time interface
class Ve_mw{
public:
     //what mouse event to do
     virtual
     void m_event()=0;

public:
     virtual
     ~Ve_mw(){}

protected:
     #ifdef Nmj_Va_mw7
     Ie_mw<Ve_mw>
                    anIe_mw;
     #endif
};
```

```
//mouse wheel up
class Te_mw_up: public Ve_mw{
public:
     //on hold
     void m_event();
};

//mouse wheel dn
class Te_mw_dn: public Ve_mw{
public:
     //on hold
     void m_event();
};
#endif

//var 4
#ifdef Nmj_Va_mw4
//mouse wheel up
class Tevent_mw_up{
public:
     //on hold
     inline
     void m_event();

protected:
     #ifdef Nmj_Va_mw7
     Ie_mw<Tevent_mw_up>
                    anIe_mw;
```

```
    #endif
};

//mouse wheel dn
class Tevent_mw_dn{
public:
    //on hold
    inline
    void m_event();

protected:
    #ifdef Nmj_Va_mw7
    Ie_mw<Tevent_mw_dn>
                anIe_mw;
    #endif
};
#endif

//var 6
#ifdef Nmj_Va_mw6
//mouse wheel
template<class Taction_mv, class Tevent>
class Ta_mwe: public Vaction{
public:
    void proceed(){ a_mw.Taction_mv::proceed(); }

    Ta_mwe():a_mw(&e_mw){}
    Ta_mwe(const Ta_mwe& obj):e_mw(obj.e_mw), a_mw(obj.a_mw,&e_mw){}
    Ta_mwe& operator=(const Ta_mwe& obj){
```

```
        if(&obj!=this){ e_mw=obj.e_mw; a_mw=obj.a_mw, a_mw.Taction_mv::set_event(&e_mw); }
        return *this;
        }

protected:
    Tevent          e_mw;
    Taction_mv      a_mw;

protected:
    #ifdef Nmj_Va_mw7
    Ia_mw<Taction_mv>
                    anIaction_mv;

    Ie_mw<Tevent>
                    anIevent;
    #endif
};
#endif

//
#ifdef Nmj_Va_mw9
template<class Tevent>
void Nmj::Tmw_p<Tevent>::proceed(){

    //timer_passed overflow workaround
    if(timer_passed < key_timer_passed){

        //just skip one m_event()
        key_timer_passed= timer_passed;
```

```
        return;
    }

    //check first press or auto repeat
    if( (timer_passed - key_timer_passed)*TIMER_MS >= Va_MOUSE_WHEEL_REP_MS   ){

        key_timer_passed= timer_passed;
        if(do_event)do_event->m_event();
    }
}
#endif

#ifdef Nmj_Va_mw4
inline void   Nmj::Tevent_mw_up::m_event(){ mouse_event( MOUSEEVENTF_WHEEL, 0, 0, WHEEL_DELTA,
0 ); }
inline void   Nmj::Tevent_mw_dn::m_event(){ mouse_event( MOUSEEVENTF_WHEEL, 0, 0,
-WHEEL_DELTA, 0 ); }
#endif

//
#ifdef Nmj_Va_mw9
void                    Nmj::Ta_mw::proceed(){ mw_p.Tproceed::proceed(); }
Nmj::Ta_mw::Vevent*         Nmj::Ta_mw::get_event()const{ return mw_p.do_event; }
void                    Nmj::Ta_mw::set_event(Vevent *const e){ mw_p.do_event=e; }
#endif

#ifdef Nmj_Va_mw2
namespace{
Tevent_mw_up e_mw_up;
```

```
Tevent_mw_dn e_mw_dn;
}
void Nmj::Te_mw_up::m_event(){ e_mw_up.m_event(); }
void Nmj::Te_mw_dn::m_event(){ e_mw_dn.m_event(); }
#endif
```

Торжество разума.

8. И наконец самое интересное.

Как мог бы выглядеть C++, если бы имел встроенную проверку интерфейсов времени компиляции?
Написал сразу на английском, чтобы два раза не писать.

```
the code above is C++ "allows but does not support" low level implicit way to use the following proposed rules:

compile time interface for abstract types of data (ADT)
#//mouse wheel event
#using interface = std::ct_adt_interface;
#interface Ia_mw_event{
#public:
#     void  m_event();
#};
';' is mandatory here, the same as class scope

interface name can be used instead of typeless "class" keyword for template parameter declaration
interface name can be used to specialization of declared typeless "class" template parameter
#//mouse wheel
#template<Ia_mw_event Tevent>
#class Ta_mw: public Vaction{
#public:
#...

interface can not be used as run-time template (as ordinary C++ run-time interface "class" does)
because there is no the object in memory as "interface" and object of "interface type" can not be defined
```

```
abstract types of data itself has no "inheritance" mechanism of class creation of object programming
so interface can not be inherited as ordinary C++ class does
and interface can not be inherited by class syntax to avoid confusing with class specific inheritance rules

but abstract types of data itself must have a way to generic use different ADTs with the same methods in code templates
so to declare class, which is complaint an interface, the following syntax must be used
      class name [: list of base class](optional) [interface: list of interfaces](optional) [{ class body }](class
definition);
#//mouse wheel up
#class Tevent_mw_up
#interface: public Ia_mw_event{
#public:
#...
here class Tevent_mw_up can be used where interface Ia_mw_event does

members and types which are declared in an interface must be defined in classes which are complaint the interface
and the interface members can not be removed by class declaration

to declare interface, which is complaint the other interface, the following syntax must be used
      interface name [interface: list of interfaces](optional) [{ interface body }](interface definition);
#interface Ia_mw_event2
#interface: Ia_mw_event{
#public:
#     void  m_event2();
#};
here interface Ia_mw_event2 can be used where interface Ia_mw_event does

interface can declare other interfaces as members by the following syntax of object definition of "interface type"
#interface Ia_mw_event3{
#public:
#     Ia_mw_event       a_mw_event;
#};
in classes Ia_mw_event interface type name must be replaced by class type name which are complaint the interface
here Ia_mw_event must not be used as template parameter
because all Ia_mw_event replacements are the same Ia_mw_event3 compile time interface
but template parameters used to create different compile-time interfaces by different parameters with similar compile-
```

```
time interface
in other words, all compile time interface names "already generic by default"

the same example with "template" used for another purpose and the interface definition
will produce set of different compile time interfaces Ia_mw_event4<>, depends on anIa_mw_event
#template<Ia_mw_event anIa_mw_event>
#interface Ia_mw_event4{
#public:
#     anIa_mw_event     a_mw_event;
#};

interface can define internal interfaces as own namespace members with the same generic "interface definition"

interface can define own members by exporting member declaration from other interfaces with "using" by the following
syntax
      using [interface](optional) interface_name;
      using interface_name::member_name;
in the case "new" interface is not the same as "used" though has the same members
the way is intended to break links between different classes with similar members by type, but logically not equal ones
interface can explicitly redeclare items inserted by "using" by ordinary rules

interface can define own scope member by the following syntax of namespace definition
      namespace scope_name{
      }
"interface scope" can be accessed with "::" operator like this: "interface_name::scope_name", not by "."
interface scope can define own members by exporting member declaration from other interfaces with "using" keyword
interface can explicitly redeclare items within namespace inserted by "using" by ordinary rules

class can include interfaces with "using" keyword for internal purposes
all members of the "using" interface must be declared in the class
but the class will not be complaint the "using" interface

changed interface can be added to class outside of class definition by ordinary work with interfaces
which are already declared inside class definition
(work by inherinace existed interface, by include existed interface, etc)
in general case it is prohibited to add changed interfaces to class outside of class definition
```

```
so in order to add new interfaces to class outside of class definition
in general case you must create new class but the same as existing
but to avoid the useless work with run-time properties of the existing class
there is a way to add new interfaces to class outside of class definition
by the following syntax
      interface interface_name = class_name [, class_name](optional);
"interface_name" and "class_name" must be declared (but may be not defined)
"class_name" can be tested for compliant to "interface_name" in the point of both class and interface definition
the same as if "interface_name" was declared inside "class_name" definition
```

===
Конец текста