# Advanced C++

Ron Akkersdijk

Come along. Saxion.

Autumn 2014

SAXION

# Literature


- Tomasz Müldner, *Programming with Design patterns Revealed*, ISBN 0-201-72231-3, Addison Wesley,
- Bjarne Stroustrup, *De programmeertaal C++*, ISBN 90-430-0231-3, Addison Wesley
- Andrew Koenig, Barbara E. Moo, *Accelerated C++, Practical Programming by Example*, ISBN 0-201-70353-X
- John Lakos, *Large Scale C++ Software*, Addison Wesley.
- Stanley B. Lippman, Josée Lajoie, *C++ Primer*, Third Edition, ISBN 0-201-82470-1, Addison Wesley Design, ISBN 0-201-63362-0, Addison Wesley

# Various Resources

Documentation:

- http://en.cppreference.com/w/
- http://www.parashift.com/c++-faq-lite/

Tools:

- http://www.codeblocks.org/
- http://cppcheck.sourceforge.net/
- http://valgrind.org/
- http://www.stack.nl/~dimitri/doxygen/

# Overview

- Internal and External Linkage
- Callbacks & Function/Method Pointers
- Template Classes & Functions
- Generic Algoritms
- Smart Pointers
- Calling C++ from C
- Reflection in C++
- The new C11++ standard

# Declaration & Definition

- A Declaration:
  - Informs the compiler about something
  - Has no side-effects
  - May be done multiple times
- A Definition:
  - The actual thing
  - Has side-effects
  - May be done only once
- The C++ rule: “declare before use”

# Definition ≡ Declaration

- A definition also serves as a declaration unless it:
  - declares a method or function without it's body
  - declares a static class data member ("attribute") within a class definition
- In both cases the actual definition occurs elsewhere

# Declaration ≡ Definition

- A declaration also serves as a definition unless it:
  - is a typedef
  - only declares the name of a class without it's definition
  - declares a method or function without it's body
  - declares a static class data member ("attribute") within a class definition
  - has an extern specifier without initialisation or function body

# Declaration Examples

```
class Employee;     // there exists a class ...
class Employee;     // it is oke to repeat this
friend Employee;    // oke, Employee was a class
friend class Employee;   // two declarations in one
int function(int, int); // there exists a function ...
int function(int, int); // repetition oke
typedef unsigned size; // 'size' is a new pseudo-type
typedef unsigned size;  // it's getting boring
extern int aGlobalVariable; // an int "somewhere"
extern int aGlobalVariable; // groan
```

# Definition Examples

```
class Queue { … };
struct filesystem { … };
enum Color { Red, Green, Blue };
template<typename T>
   void sort(const T *a[], int n ) { … }
int anInteger; // not extern
extern int anotherInteger = 1; // has initialiser!
int function(int i, int j) { … }
static int staticFunction(int i) { … }
inline int inlineFunction(int i) { … }
```

# Linkage

- To link = to connect the names
  (aka the “symbols”)
  to what they represent
- Two forms:
  - Internal (within a single .o file)
  - External (across multiple .o files)
- <u>Declarations</u> inform the compiler about things that may exists in another file, or later in this source file

# Internal Linkage

- Sometimes linkage can only be internal:
- In that case a .o file will not contain any visible information about them
  - enum en typedef definitions
  - class/struct/union definitions
  - inline (member-)functions
  - template definitions
  - static global functions & global objects (this is not about static members!).
- Only the first 63 characters of the name make it unique within that source file

# External Linkage

- If a name refers to something in another file then linkage is external
- Both .o files must contain information about that name (i.e. “used” and “defined”)
- Only the first 31 chars of the name count

# Name Mangling

- Most C++ compilers do some hidden magic with names, for instance
  - **Student::isIBG(int)**
  - **Doosje<int>::Doosje(int&)**
- becomes something like
  - **__ZN7Student5isIBGEi**
  - **__ZN6DoosjeIiEC1ERi**
- so beware of external linkage,
  that 31 character limit is reached soon

# Linkage

- Q: Why 63 and 31 characters ?
  Why not 64 and 32 ?
  - A: The compiler adds an extra '_' in front of the symbol to prevent accidental conflicts with names from code written in assembler
- Q: Are those limits absolute?
  - A: No, some compilers/platforms support more
  - A: It is simply the C++ standard's <u>minimum</u>

# Questions?

# Callbacks

- Definition: A callback is a function or method provided by a client to some subsystem ...
  - ... so that that subsystem can perform an operation in the context of the original client
  - ... without having to know all the implementation details of that client
  - is an example of *Separation of Concerns*
- Inheritance and virtual functies can be used to realise a type-safe callback mechanism

# Comparable

```
#ifndef COMPARABLE_H
#define COMPARABLE_H

class Comparable { // Java "interface"
public:
   // the "mandated" callback method
   virtual int compare(const Comparable*)
                    = 0;    // a pure-virtual method
};

#endif /*COMPARABLE_H*/
```

# Comparing Fruit

```
class Fruit : public Comparable {
  const string     name;
public:
  // a possible implementation
  int compare(const Comparable *cp) {
    require(cp != 0); // not null
    const Fruit *fp
          = dynamic_cast<const Fruit*>(cp);
    require(fp != 0); // real type matches
    return name.size() - fp->name.size();
  }
};
```

# A Sorting class

```
#ifndef SORTER_H
#define SORTER_H
#include "Comparable.h"
class Sorter  {
public:
   static void quicksort(
                   const Comparable *a[]
                   , int low, int high );
private:
   static void swap(const Comparable *a[]
                   , int i, int j );
};
#endif
```

# A Sorting class

```
void Sorter::quicksort(const Comparable *a[]
                               , int low, int high) {
  if (low < high) { // to stop recursion
    int  lo = low,  hi = high + 1;
    const Comparable  *elem = a[low];
    for (;;) {
      while (++lo <= high && a[lo]->compare(elem) < 0)
          ;
      while (a[--hi]->compare(elem) > 0)
          ;
      if ( lo < hi )
          ...
```

# A Sorting class

```
        …
        if ( lo < hi )
           swap (a, lo, hi);
        else
           break;
     } // end of for(;;)
     swap(a, low, hi);
     quicksort(a, low, hi - 1);
     quicksort(a, hi + 1, high);
  }
}
```

# The application

```
int main() {
   Fruit  *fruit[3];
   fruit[0] = new Fruit("bananas");
   fruit[1] = new Fruit("cherries");
   fruit[2] = new Fruit("apples");

   Sorter::quicksort( (Comparable*[]) fruit, 0, 2 );

   for (unsigned i = 0; i < 3; ++i)
        cout << *(fruit[i]) << endl;
}
```

# Interfaces and C++

- To get the C++ equivalent of java interfaces:
  - define (abstract) classes with pure virtual callback methods

```
class Comparable { … };
class Serializable { … };
```

  - and use multiple inheritance

```
class Fruit : public Vegetable // normal baseclass first
            , public Comparable, public Serializable, …
{
   …
};
```

# Questions?

# Pointers to functions

- Why use them:
  - Can easily be retro-fitted i.e. no need to add callback methods to classes later
  - The function name is not fixed, so it is more flexible (e.g. sort_by_name, sort_by_age)
  - Many algorithms in the standard template library expect them (or function objects)
  - Many existing libraries use it

# Pointers to functions

- Let start with:

```
void quicksort(int[],int,int); // a function
```

- The definition of a pointer to that function:

```
void (*pf) (int[],int,int) = quicksort;
   // just the name of a function gives it's address
```

- Then call the function using:

```
 (*pf)(a, b, c);     // via dereference operation
```

- or, shorter:

```
pf(a, b, c);           // like an array: a[i]
```

# Pointers to functions

- Caveat: Don't get

```
void (*pf)(int[],int,int); // two pairs of ()
```

- mixed up with:

```
void *pf2(int[],int,int);  // one pair of ()
```

- which means that pf2 is a function, returning a pointer to an unknown type
- while pf is a pointer to a function, returning an unknown type (i.e. nothing)

SAXION

# Pointers to functions

- We can now define an array of pointers:

```
void (*sortfuncs[])(int[],int,int) = {
        quicksort,      mergesort,
        heapsort,       bubblesort
};
```

- And then call e.g. “quicksort” as:

```
(*sortfuncs[0])(.....);
```

- Or, shorter, as:

```
sortfuncs[0](.....);
```

# Pointers to functions

- Let's define a pointer to that array:

```
void (**pfsort)(int[],int,int) = sortfuncs;
```

- And then call e.g. “quicksort” as:

```
(*pfsort[0])(.....);
```

- Or, again shorter, as:

```
pfsort[0](.....);
```

# Example: _new_handler

- The C++ library contains a variable:

```
void (*_new_handler)();
```

- which will be called when operator new fails and _new_handler != 0
- Can be set via

```
_new_handler = myFreeStoreException;
```

- of via

```
set_new_handler(myFreeStoreException);
```

# As parameter

- A “tuneable” sort function ...

```
void sort(int data[], int low, int high,
          void (*method)(int[],int,int) = quicksort)
{         // the 4th parameter determines "how"
   require(data != 0);
   require(method != 0);
   (*method)(data, low, high);
}
```

- And then use it like:

```
sort(ia, 0, iasize, bubblesort);
sort(ia, 0, iasize); // using quicksort
```

SAXION

# Example: list<T>

- A list<T> container can be sorted
- By default it uses the '<' operator

```
list<string> l;
l.sort(); // lexicographic sort
```

- You can also provide a “<” function

```
bool byLength(const string& a, const string& b)
{
  return a.size() < b.size();
}
l.sort(byLength);  // sorts short to long
```

# As return values

```
int ( * ff(int) ) (int[],int);
```

- Meaning: ff() is a function, with one int parameter, and the return value is a pointer to a function of type:

```
int (*) (int[],int);
```

- Using typedef to improve readability gives

```
typedef int (*PIF)(int[],int);


  PIF  ff(int);
```

# Example: signal

- The Unix systemcall

```
void (*signal(int,void(*)(int)))(int);
```

- After

```
typedef void (*SIG_FUN)(int);
```

- a bit more readable

```
SIG_FUN  signal( int, SIG_FUN );
```

# DIY excercises

```
typedef int (*PIF)(int,int);
int a (int,int);
int * b (int,int);
int (* c) (int,int); //PIF c;
int d (int, int (*)(int,int));  //int d (int, PIF);
int * e(int, int (*)(int,int)); //int *e (int, PIF);
int (*f)(int,int (*)(int,int)); //int (*f) (int, PIF);
int ( * g (int,int) ) (int, int); //PIF g (int,int);
int ( * h (int, int (*) (int,int))) (int,int);
     // PIF h (int, PIF);
int ( * ( * i )( int, int(*)(int,int)))(int,int);
     // PIF ( * i ) ( int, PIF);
```

# Questions?

# Pointers to member functions

- Always related to some class type:

```
class Screen {
   int    getHeight() { return height; }
};
int (*pfi) ();              // ordinary function pointer
pfi  = & Screen::getHeight; // error, type mismatch
int (Screen::*pmi) ();      // Screen-method pointer
pmi  = & Screen::getHeight;   // oke
// Beware: The '&' is mandatory here !
```

- Exception: static methods are treated like ordinary functions!

# The “.*” and “->*” operators

```
Screen  myScreen, *bufScreen = &myScreen;

// the direct invocation of a member function
if ( myScreen.getHeight() == bufScreen->getHeight() )
     bufScreen->copy( myScreen );

// the equivalent using pointers to members
int  (Screen::* pmi)()  = & Screen::getHeight;
void  (Screen::* pmv)(Screen&) = & Screen::copy;
if ( (myScreen.*pmi)() == (bufScreen->*pmi)() )
     (bufScreen->*pmv)( myScreen );
```

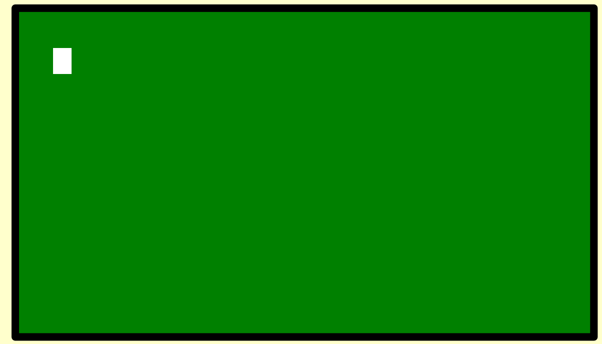
# Using typedef

```
typedef  Screen &  (Screen::* Action) ();
             // Actually a ScreenAction  8-)

class Screen {
public:
   Screen &    forward();
   Screen &    down();
   // ....
   Screen &    repeat( Action = &Screen::forward,
                       unsigned = 1 );
};
```

# Define and use 'repeat'

```
Screen&  Screen::repeat(Action op, unsigned times) {
  for ( unsigned  i = 0; i < times; ++i )
     (this->*op)(); // invokes some Screen method
  return *this; // finally return self
}

Screen  myScreen;  // a Screen instance

  myScreen.repeat( &Screen::down, 20 );
  myScreen.repeat();
  // by default: repeat( &Screen::forward, 1)
  myScreen.repeat().repeat(); // "chaining" actions
```

# An array with Actions



```
Action  menu[] = {
   & Screen::home,
   & Screen::forward,
   & Screen::back,
   & Screen::up,
   & Screen::down,
   & Screen::bottom
};
enum CursorMovement = { HOME, FORWARD,
                        BACK, UP, DOWN, BOTTOM };
Screen& Screen::move(CursorMovement cm) {
   (this->*menu[ cm ])(); // call the matching method
   return *this;
}
```

# A generic menu handler

- In non-GUI applications users often need some kind of menu.
- Each item in the menu often corresponds to a single use-case.
- Instead of writing yet another menu handler for each application it makes more sense to write a generic solution.
- All that than remains is to specify the actual methods to execute.

# MenuHandler class

```
class Application;    // forward declaration
class MenuFunction;   // forward declaration

class MenuHandler {   // The generic handler
  Application             * const     appl;
  vector<MenuFunction*>  const   &   functions;
public:
  MenuHandler(Application *ap)
     : appl(ap)
     , functions(ap->getFunctions()) {
  }
  void  showMenu() const;
};
```

# Application baseclass

```
class Application {     // The Application baseclass
protected:
   const string            description;  // menu title
   vector<MenuFunction*> functions;    // menu entries
public:
   Application(const string& s) : description(s) {}
   virtual ~Application() { … } // for cleanup
   const string&    getDescription() const
          { return description; }
   const vector<MenuFunction*>&  getFunctions() const
          { return functions; }
};
```

# MenuFunction class

```
// The pseudo type for an Application method
typedef void (Application::* ApplFunction) ();

class MenuFunction {
private:
   const string           description;
   const ApplFunction     function;
public:
   MenuFunction(...,...) : ... {}
   const string           getDescription() const {...}
   const ApplFunction     getFunction() const {...}
};
```

# MenuHandler::showMenu()

```
void MenuHandler::showMenu() const {
   for (;;) {
      // print heading
      cout << "\n\tTUI: "
            << appl->getDescription() << endl;
      // print the menu
      for (unsigned i = 0; i < functions.size(); ++i) {
         cout << "\t" << (i+1) << "\t"
               << functions[i]->getDescription()
               << endl;
      }
      cout << "\t0\texit" << endl;
      ...
```

# MenuHandler::showMenu()

```
    ...
    cout << "\t\tChoose action: " << flush;
    unsigned  chosenindex = 0;
    cin >> chosenindex;    // read choice
    if (chosenindex == 0)
       return;
    if ( (chosenindex >= 1)
     &&  (chosenindex <= functions.size()) )
    {    // call the chosen method
       (appl->*(
          functions[chosenindex-1]->getFunction())) ();
    } else
       cout << "sorry, no such function" << endl;
  } // end forever
}
```

# A Factory example

```
// A derived class for a specific Factory
class Factory : public Application {
   ...
public:
   Factory();
   // the methods to be called from the menu
   void addSupplier();
   void addMachine();
   void addProblem();
   ...
};
```

# The Factory constructor

```
#define METHOD(method) \
    static_cast<AdminFunction>(&Factory::method)

Factory::Factory()
  : Application("Factory Administration")
{
    // register some methods as menu-functions
    menufunctions.push_back(
        new MenuFunction( "add a supplier",
                    METHOD(addSupplier)) );
    menufunctions.push_back(
        new MenuFunction( "add a machine",
                    METHOD(addMachine)) );
    menufunctions.push_back(
        new MenuFunction( "report a problem",
                    METHOD(addProblem)) );
}
```

# The main function

```
int main()
{
   Application  *appl = new Factory();
   MenuHandler  *handler = new MenuHandler(appl);
   handler->showMenu();
   delete handler;
   delete appl;
}
```

# Questions?

# Pointers to member functions

- Q: Why can't I mix function/methods pointers?
- A: Because methods have a secret “this” parameter, so ...

```
class Screen {
   int     getHeight( … );
};
```

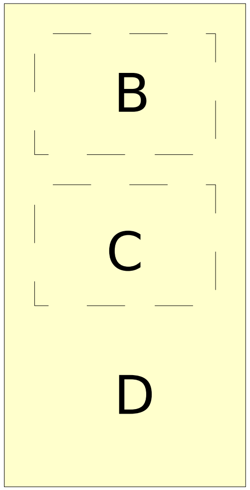
- Actually compiles as:

```
int  Screen::getHeight( const Screen *this, … );
```

# Multiple inheritance pitfalls

- In the previous example we, safely, “up-casted” a Factory method pointer to a Application method pointer
- When using multiple inheritance up-casting can be dangerous!

```
class D : public B, public C {
   int    method( … );
};
```




- The D and B “views” have the same “this” addres, the C “view” does NOT, because it exist after the B part.
- So the C part has a different “this” value, but static_cast<…> ignores that difference!

# C++ & virtual methods

- C++ itself uses “pointer to method” for virtual methods!

```
class Base {
   virtual void f();   // 1e virtual
   virtual ~Base();    // 2e virtual
   virtual int g();    // 3e virtual
};

class Derived : public Base {
   ~Derived();         // 2e virtual
   void f();           // 1e virtual
};
```

# C++ & virtual methods

- Secret “RunTimeTypeInformation”

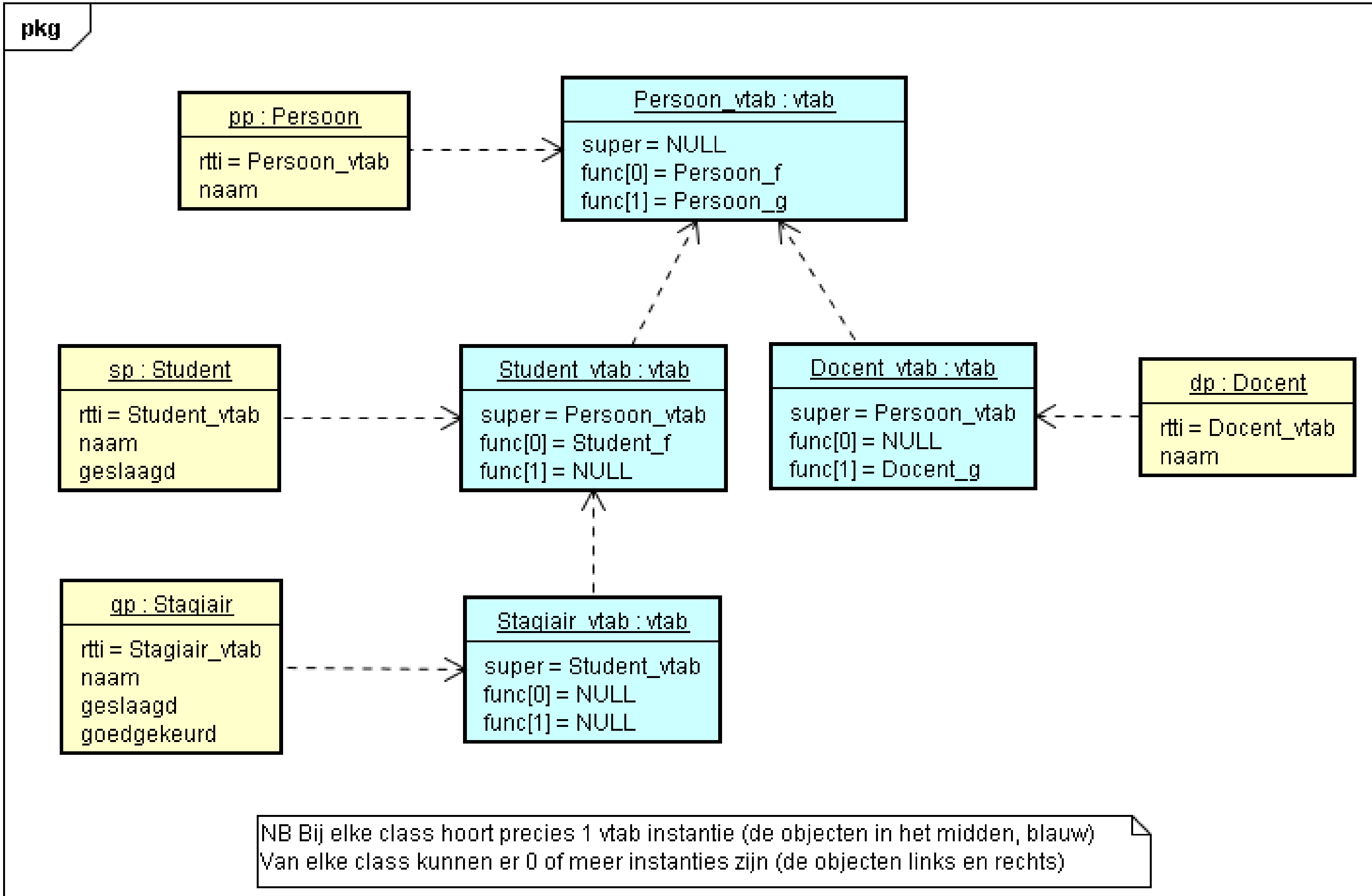
```
class vtab {     // "virtual method table"
   vtab*   super;      // to vtab of base class or 0
   void  (*func[])(); // method pointer array
};
// The vtab for class Base
vtab  Base_vtab = { 0, // 0=no supper class
     { &Base::f, &Base::~Base, &Base::g } };

// The vtab for class Derived
vtab  Derived_vtab = { &Base_vtab, // "from Base"
     { &Derived::f, &Derived::~Derived, 0 } };
```

no Derived::g

# C++ & virtual methods

- Secret “RunTimeTypeInformation”

```
class Base {
   vtab*    rtti;        // the secret vtab pointer!
   // initialization code added to constructor
   Base() : rtti(&Base_vtab), … // initializers
};
class Derived {          // re-initializes rtti
   Derived() : Base(), rtti(&Derived_vtab), …
};
   // a virtual method call will behave as
   (rtti->func[number])( this, … )
   // while searching the inheritance tree
   // upwards for a matching method
```


pkg
pp : Persoon
rtti = Persoon_vtab
naam
Persoon_vtab : vtab
super = NULL
func[0] = Persoon_f
func[1] = Persoon_g
sp : Student
rtti = Student_vtab
naam
geslaagd
Student_vtab : vtab
super = Persoon_vtab
func[0] = Student_f
func[1] = NULL
Docent_vtab : vtab
super = Persoon_vtab
func[0] = NULL
func[1] = Docent_g
dp : Docent
rtti = Docent_vtab
naam
gp : Stagiair
rtti = Stagiair_vtab
naam
geslaagd
goedgekeurd
Stagiair_vtab : vtab
super = Student_vtab
func[0] = NULL
func[1] = NULL
NB Bij elke class hoort precies 1 vtab instantie (de objecten in het midden, blauw)
Van elke class kunnen er 0 of meer instanties zijn (de objecten links en rechts)

# C++ & virtual methods

- Secret “RunTimeTypeInformation”

```
class Base {
   vtab*     rtti;        // the secret vtab pointer!
   ...
};
vtab   Base_vtab ...;        // vtab for class Base
vtab   Derived_vtab ...;     // vtab for class Derived
```

- dynamic_cast<SomeClass*>(...) uses this information to determine the true type of an object: “*Does the rtti pointer equals &SomeClass_vtab or the vtab for a derived class*”

# Questions ?