Chapter 6

Object-Oriented Techniques and Numerics

Object oriented techniques have proven to be a good tool for software design and development. The first object-oriented programming language is *Simula-67*. It has been designed by Dahl, Myhrhaug and Nygaard at the Norwegian Computing Center in Oslo [25]. Today there are many object-oriented programming languages. Two other popular examples are *Smalltalk* and *C++* . *C++* was introduced by Stroustrup in 1985[26] and it has been strongly inspired by Simula. The current state of the *C++* language is described in the third edition of Stroustrup's popular book [27]. Although object oriented software design has been used for a long time, it is only now becoming popular in the field of numerical simulations. One reason is the bad reputation object oriented languages had in terms of computational efficiency. With the availability of parameterized types (templates) in *C++* , object-oriented techniques become more and more accepted in the numerical community. Templates enable the programmer to use a clean and object-oriented design without the performance problems experienced so far. This chapter tries to first briefly introduce the object oriented paradigm. The introduction intends to point out the ideas of object oriented programming. It is not meant to be a comprehensive explanation of object oriented techniques. More details can be found in the literature, for example in [28]. The application of those techniques to finite-volume simulation software shall be sketched. An obvious design, as it would be natural following the object oriented paradigm will be presented first. This will, however, lead to performance problems. Hence it is necessary to find a good compromise between good design and high efficiency. There are a couple of recent developments which allow this performance problem to be overcome. A good overview of different techniques for object-oriented numerics, using *C++* , can be found in [29]. Templates, as already mentioned above, had a very strong influence on the design of numerical software. Some of the techniques which can be used to overcome efficiency drawbacks shall be described in brief. It is not intended to go into details of the actual design used for the *MOUSE* library. These details can be found in chapter 7. The introduction is, however, a little biased towards the *C++* language, because this is the programming language which has been used to code the finite volume library described later.

6.1 Brief Introduction

Object-oriented design models things (objects) from the real world with appropriate computer-understandable descriptions. Of course not everything what is described as a computer object corresponds to a real world object. If you think of a window on the screen, it is certainly not related to any real world object. It is, however, very well described as an object in terms of object-oriented programming. In the virtual world of the computer it is an object and it corresponds with other objects in the system. One of the major differences of object-oriented programming versus classical programming is the unification of data structures and functionality. Consider, as an example, graphical primitives of a drawing program. This could be circles, lines, squares or other graphical objects. All these objects have things in common. For instance they have a position, and they can be moved. A classical program would define appropriate data structures to store those objects. Then functions could be defined to operate on the data structures and for example move a circle from one position to another. In an object-oriented approach data and functionality would be combined. This means a graphical object might have a data-structure to store its position. Furthermore it does 'know' how to move itself. Data which is stored within an object is often referred to as *attributes*, whereas a functionality is called an *operation.*

6.2 Classes and Inheritance

Objects can be subdivided into several different classes of objects. This classes can be refined and more specialized classes of objects can be defined. Consider as an example fruit. Apples, perches, peaches, mangos and a lot more are objects. All these objects are fruits and hence belong to the class fruit. They all have properties in common, they are eatable, they contain a certain amount of sugar and they need a certain time to ripe. There are of course other properties they might share. It is now possible to refine the class fruit. For instance it would be possible to talk about tropical fruit, northern fruit and maybe fruit which grow everywhere. These classes of fruit can be even more specifically described. Bananas, Mangoes, melons and others are all tropical fruit. Still, there are different kinds of melons, which all belong to the class melon. All melons, however, are tropical fruit and of course they are fruit. These classification of objects is a natural and very convenient way to analyze objects from the real world. The more general the class used to describe an object is, the more abstraction has been used to describe it. There is, of course, not one unique and correct way to describe a system. It is a huge part of the development to find the right abstractions for a system. Instead of dividing the class fruit into tropical and northern fruit, it might as well be useful to talk about African, Asian, European, American or Australian fruit.
Another classification example are vehicles. Figure 6.1 illustrates the classification of vehicles. Of course this is only one possible classification. It also only is a computer model. Such a model will never contain every aspect of the real thing. To find proper abstractions is a very important part of object-oriented design. The process of refining the classification

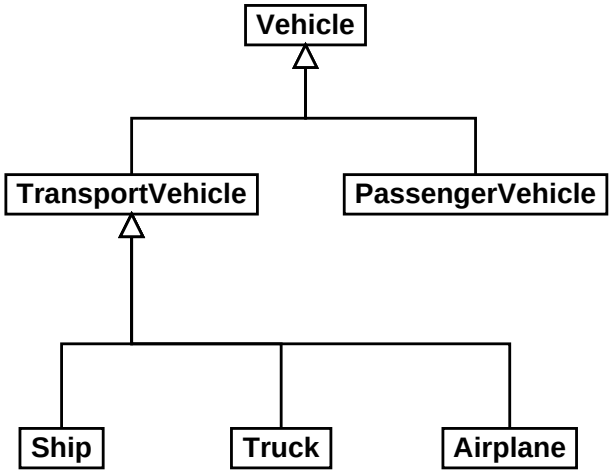

Figure 6.1: Vehicles classification.

is also called inheritance. Every `Vehicle` has certain properties. For example it can move and it has a position. If now a new class, for instance `TransportVehicle`, is derived from `Vehicle` it inherits all properties of its base class. Furthermore a `TransportVehicle` will have a payload and it can load or unload.

6.3 Encapsulation

A very important issue of object-oriented programming is data encapsulation. An important goal of software design is to keep interfaces clean and to minimize the potential to introduce errors in a code. This becomes particularly important in larger projects where many individuals work on different parts of one code. It is also crucial for development libraries. The following small example intends to illustrate data encapsulation. Consider a **struct** in traditional *C* (see listing 6.1). The **struct** `item_t` can be used to realize a linked list of floating point values in the memory. The pointer `next` defines the next item in the memory. If it becomes a zero-pointer the list has ended. Now a complete list can be accessed by knowing its first element. Figure 6.2 illustrates this. The function `printall` (see listing 6.1) will print all values on the screen. If used on the example shown in figure

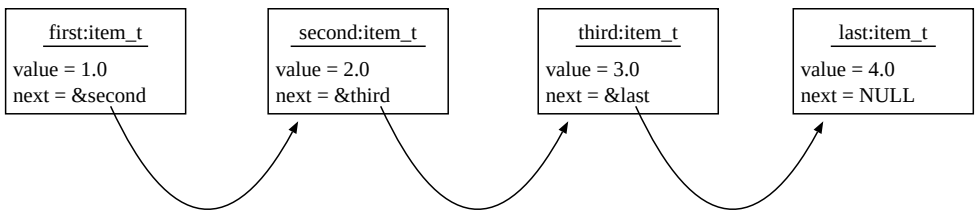

Figure 6.2: A linked list.

```
   typedef struct item_t
2  {
     item_t *next;
4    double value;
   };
6
   void printall(item_t *item)
8  {
     do {
10     if (item->next == NULL) {
         printf("%4.2f \n", item->value);
12     } else {
         printf("%4.2f ", item->value);
14     };
       item = item->next;
16   } while (item != NULL);
   };
```

Listing 6.1: A linked list in *C* .

6.2 it will result in the following screen output `1.00 2.00 3.00 4.00` This example can be seen as four objects which form a list. Each object has one *attribute*, namely the floating point value `value`. It also has an *operation*, although it has not been explicitly coded as a function in *C*. All objects from this example have the *operation* 'get next'. The actual use of this *operation* in a *C* program is, however, a simple pointer assignment. The implementation of a linked list shown here has one major drawback. Assume in line 10 of listing 6.1 the comparison operator `==` would have been mistaken with the assignment operator `=`. So, that line four of listing 6.1 becomes `if (item->next = NULL) {`. The pointer `item->next` will then have the value `NULL`, which will lead to unpredictable program behavior. This sort of memory bugs are very hard to find. It will become especially painful if it occurs in a part of a large project, which has been written by another person. Therefore it seems to be essential to provide a strict interface, which does not allow the underlying data structures to be damaged by 'user' programmers. A simple link-entry realized in the *C++* programming language can be found in listing 6.2. Now the *operation* `getNext` has been explicitly realized as a function within the class `Link`. In terms of object-oriented programming `getNext()` is called a method of `Link`. The `value` field of `Link` has been declared to be private. This means it can only be accessed or modified from methods of the class `Link`. By doing so it has been ensured that the linked list cannot be damaged by 'user' programmers. Now a similar mistyping as above `if (item->getNext() = NULL)` would result in a compiler error and not in unpredictable program behavior. This method of hiding information and providing access methods is called *encapsulation*. The advantage of preventing internal structures from damage is very important in larger software environments.

```
class Link
{
private:
  Link *next;
public:
  Link* getNext() { return next; };
  void insert(Link *link) {
    link->next = next;
    next = link;
  };
  double value;
};

void printall(item_t *item)
{
  do {
    if (item->getNext() == NULL) {
      cout << item->value << ' ';
    } else {
      cout << item->value << endl;
    };
    item = item->next;
  } while (item != NULL);
};
```

Listing 6.2: A linked list in *C++* .

6.4 Polymorphism

Polymorphism is the ability of an *operation* to have different forms in different classes. Going back to the *Vehicle*-example, any *Vehicle* can move from one *Position* to another. Figure 6.3 shows a more detailed representation of this example. The following pseudo code example (see listing 6.3) shows how a *Cargo* can be shipped no matter which type of *TransportVehicle* is used. The *operations* `load`, `move` and `unload` are called *virtual functions* in *C++* terminology. The basic idea is to describe an abstract interface, which can be relied on. In this example every *Vehicle* has a `move` method. What actually happens within this method is described in more specialized derived classes. For instance

```
void ship(Cargo cargo,
          TransportVehicle vehicle,
          Position destination)
{
  vehicle.load(cargo);
  vehicle.move(destination);
  vehicle.unload();
};
```

Listing 6.3: `TransportVehicle` used as an abstract argument.

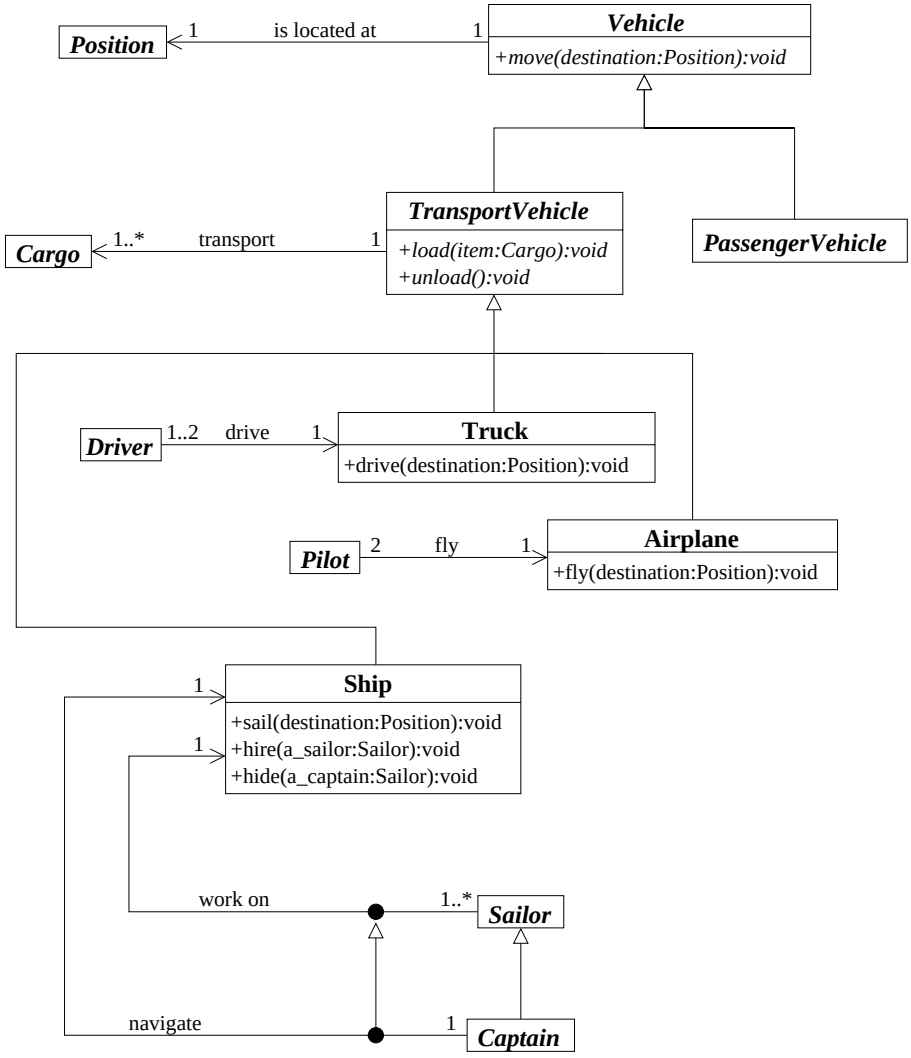

Figure 6.3: Vehicles classification.

would the `move` method of an *Airplane* call its `fly` method. If such a *virtual* method is called, the actual jump-position can only be resolved during run-time, not compile-time. This might lead to serious performance problems if a method is called very often and computes relatively little. These issues shall be addressed in detail in the next section.

6.5 A Simple Numerical Example

This section tries to show the usage of object-oriented techniques together with a small numerical example. Any one dimensional function $f(x)$ shall be integrated in an interval $x \in [x_1, x_2]$. Different approaches will be shown here and they will be compared in terms of computational efficiency. Different functions will be used to test the implementations:

$$f_m(x) = x^m \tag{6.1}$$

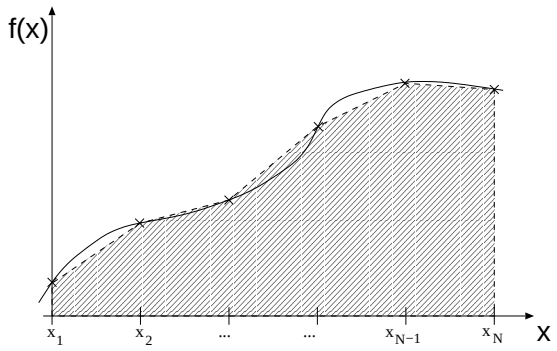

Figure 6.4: Numerical integration.

Numerical integration, using N_{nds} sample points $\{(x_1, f(x_1)), \cdots, (x_{N_{\text{nds}}}), f(x_{N_{\text{nds}}}))\}$ is performed using a simple formulation (6.2):

$$\left[\int_{x_1}^{x_2} f(x)\mathrm{d}x\right] = \frac{1}{2}\sum_{i=1}^{N_{\text{nds}}-1}\left((x_{i+1}-x_i)(f(x_i)+f(x_{i+1}))\right). \tag{6.2}$$

Figure 6.4 illustrates this. The hatched region is the area computed by the integration formulation used. $x_{i+1}-x_i$ shall be constant in this example, so that the above formulation can be optimized for computation in the following way:

$$\left[\int_{x_1}^{x_2} f(x)dx\right] = \Delta x\left(\frac{f(x_1)+f(x_n)}{2}+\sum_{i=1}^{N_{\text{nds}}-1} f(x_i)\right). \tag{6.3}$$

A small *C* program has been used as a reference (see listing 6.4). Where `FUNC` is a macro and has to be defined before the `integrate` function. For $f(x) = x^3$ it could look like this: `#define FUNC(x)(x*x*x)`
Different object-oriented approaches shall be compared now.

```
double integrate(double x1, double x2, int n)
{
  double x, delta_x, sum;
  int i;
  delta_x = (x2−x1)/(n−1);
  sum = 0.5∗(FUNC(x1)+FUNC(x2));
  for (i = 2, x = x1 + delta_x; i < n; ++i, x += delta_x) {
    sum += FUNC(x);
  };
  return delta_x∗sum;
};
```

Listing 6.4: Simple integration function in *C* .

```
class Function
{
public:
  virtual double operator()(double x) = 0;
};

class Xpow3 : public Function
{
public:
  virtual double operator()(double x) { return x*x*x; };
};

double integrate(Function &f, double x1, double x2, int n)
{
  double x, delta_x, sum;
  int i;
  delta_x = (x2-x1)/(n-1);
  sum = 0.5*(f(x1)+f(x2));
  for (i = 2, x = x1 + delta_x; i < n; i++, x += delta_x) {
    sum += f(x);
  };
  return delta_x*sum;
};

int main()
{
  Xpow3 f;
  double y = f(2.0);
  double F = integrate(f , 0, 1, 100);
};
```

Listing 6.5: Integration using a virtual method.

6.5.1 Dynamic Polymorphism (Virtual Function)

Listing 6.5 shows a *C++* version of the integration problem. This example is using dynamic polymorphism. An abstract **class** `Function` (see listing 6.5, line 1) is created, which then is used as argument for the `integrate` routine (see listing 6.5, line 13). The `= 0` indicates that the parenthesis operator has to be overwritten in a derived class to be actually used. As an example of a derived class, consider the implementation of $f(x) = x^3$. The `integrate` function, taking a `Function` as parameter will only slightly change. The statement in line 29 of listing 6.5 will compute the integral $\int_0^{+1} x^3 dx$ This will compute the integral, using 100 discrete points. Unfortunately this implementation turns out to have at most 15-20% of the performance of the original simple C function. This performance penalty can be explained by the runtime resolution of the jump address of `Function::operator()`. The routine `integrate(Function&, double, double, int)` can take any object, which is an instance of a derived class from `Function`, as argument. This has the advantage that only one routine is needed to integrate various functions. Furthermore different functions can be fed to this routine during run-time. Nevertheless the

```
template <class T, int DIM>
class Array
{
private:
  T value[DIM];

public:
  T& operator[](int i) { return value[i]; };
};

int main()
{
  Array<int,3> a;
  a[0] = 1; a[1] = 2; a[2] = 3;
  Array<string,2> b;
  b[0] = "a first text"; b[1] = "a second text";
};
```

Listing 6.6: A simple template example.

performance penalty is intolerable for larger problems. In this case of a small application it might not play a big role. Considering a large finite-volume framework, like the one described in the next chapter, it becomes crucial to got as much out of a computer as possible. The maintainability of a code should, however, not be too much affected by performance tuning. The next paragraph will introduce a way to overcome the performance problem, but keeping a clean and object-oriented structure.

6.5.2 Static Polymorphism (Templates)

In the last example it has been shown how a flexible routine for numerical integration could be realized in *C++* . The computational efficiency is, however, intolerably low. This is mainly due to run-time resolution of jump addresses. A closer look at the example shows, that in principle every information needed for a sufficient optimization is available during compile-time. Listing 6.7 shows a version of this integration algorithm, making use of *C++* templates. Templates enable the programmer to code algorithms, using abstract types. At first templates have been used to write generic container, which can be used to store a variety of different data. Listing 6.6 shows a small example of such a parameterized type and how it could be used. The compiler will create two different classes out of the template `Array`, namely `Array<int,3>` and `Array<string,2>`. These two classes have exactly the same structure, but they use a different data-type inside. For numerical applications a slightly different way of using templates proved to be very useful. Templates can also be used to realize polymorphic objects, similar as it has been done using a virtual method in the previous example. In listing 6.7 line 19 a template function is created. Depending on how this function is called in a program, different versions will be compiled. The function `template< class FUNC> integrate(double,double, int)` relies on the existence of `FUNC::operator()(double)`. If this function is called with a

```
struct Xpow3
{
  double operator()(double x) { return x*x*x; };
};

struct Sqrt
{
  double operator()(double x) { return sqrt(x); };
};

template <class FUNC>
double integrate(double x1, double x2, int n)
{
  FUNC f;
  double x, delta_x, sum;
  int i;
  delta_x = (x2-x1)/(n-1);
  sum = 0.5*(f(x1)+f(x2));
  for (i = 2, x = x1 + delta_x; i < n; i++, x += delta_x) {
    sum += f(x);
  };
  return delta_x*sum;
};

int main()
{
  cout << integrate<Xpow3>(0, 1, 100) << endl;
  cout << integrate<Sqrt>(2, 2.5, 50) << endl;
  cout << integrate<int>(0, 1, 100) << end; // compiler error!
};
```

Listing 6.7: Integration using a template function.

template parameter that does not have this operator, a compiler error will occur (see listing 6.7, line 29). Now different functions can be created (see listing 6.7, line 1 and 6). Please note, that a **struct** in *C++* is simply a **class**, where all fields and methods are **public**. As long as the created functions have a method **operator()(double)** they can be integrated. Very important is, that this operator does not have to be declared virtual. Now everything can be resolved during compile time and the compiler is able to effectively inline the calls to **operator()(double)**. Thus the performance is significantly higher as it is with a virtual method. This example has almost no performance loss compared with the simple *C* example. The main disadvantage of this example is, that it is not possible to select the function to integrate during run-time.

6.5.3 Combination of Static and Dynamic Binding

Here a nice approach how to combine the advantages of both implementations, presented so far, is shown. The idea is to use the same method as sketched in section 6.5.2 for

```
class Integrator
{
public:
  virtual double operator()(double x1, double x2) = 0;
};

template <class FUNC>
class TIntegrator
{
private:
  int n;  // number of sample points
  FUNC f; // function to integrate

public:
  TIntegrator(int an_n) { n = an_n; };
  virtual double operator()(double x1, double x2);
};

template <class FUNC>
double TIntegrator<FUNC>::operator()(double x1, double x2)
{
  double x, delta_x, sum;
  int i;
  delta_x = (x2-x1)/(n-1);
  sum = 0.5*(f(x1)+f(x2));
  for (i = 2, x = x1 + delta_x; i < n; i++, x += delta_x) {
    sum += f(x);
  };
  return delta_x*sum;
};

template<int EXP>
struct Pow
{
  double operator()(double x);
};

template<int EXP>
inline double Pow<EXP>::operator()(double x)
{
  double p = 1.0;
  for (int i = 0; i < EXP; i++) p*= x;
  return p;
};

int main()
{
  TIntegrator<Pow<3> > fint(100);
  cout << fint(0, 1) << endl;
};
```

Listing 6.8: Combination of templates and virtual methods.

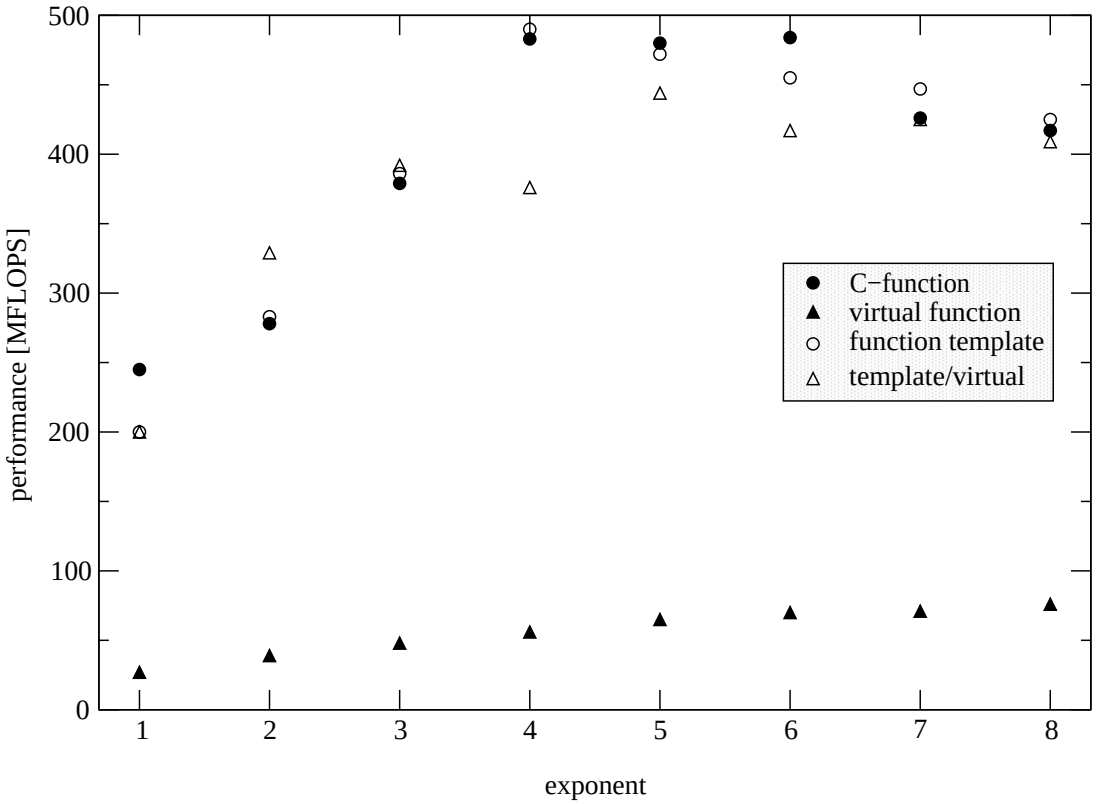

Figure 6.5: Performance of different integrate implementations.

the inner integration method. Around this inner, performance critical, part a wrapper using the virtual mechanism is created. This approach enables run-time selection of different functions together with efficient numerics. In line 1 an abstract base for function integrators is defined. The method **virtual double operator()(double,double)** shall compute the definite integral of a function.

6.5.4 Comparison

Figure 6.5 shows a comparison between the different implementations. The tests have been carried out on an *AMD Athlon 500MHz* processor. It can be clearly observed, that the pure virtual implementation is very slow compared with the others. The reason obviously is the run-time binding of the jump address. Furthermore *C++* compilers can inline function calls to completely eliminate jump statements at all. This cannot be done for virtual functions, since it is not known which version of a method will be called during run-time.