C++ for Scientific Computing

Stefan Lang

Interdisciplinary Center for Scientific Computing, University of Heidelberg

15. Oktober 2015

C++ für Wissenschaftliches Rechnen I

C++ für Wissenschaftliches Rechnen II

Requirements onto the programming language

→ Efficiency. . .

- of program
- of development

→ Hardware-related programming language

→ Integration with existing code

→ Abstraction

→

Comparison of C++ with other languages

Fortran & C

+ fast code
+ good optimization

- only procedural language
- low flexibility
- bad maintainability

C++

+ good maintainability
+ fast code
+ good integration with Fortran and C libraries
+ high degree of abstraction

- difficult to optimize
- mostly more memory consumption

Concepts of C++

C++ is an object-oriented language

this means C++ supports

1. Abstraction by classes and objects,
2. Inheritance and
3. Polymorphism during runtime.

Polymorphism means „Many Shapes“:

- A variable can change its type during runtime,
- A function with polymorphic arguments,
- A function name, that is used by functions with different impelementation.

Literature

Literature for C++

- B. Stroustrup: C++ – The Programming Language (The Bible)
- B. Eckel: Thinking in C++, Volume 1 + 2
- A. Willms: C++ Programmierung (well for beginners!)

Basic C++ Knowledge

To exhaust the advantages of C++ abstract techniques are necessary. The following basic concepts are as a basis imperative:

- Basic data types and control structures:
  - `int`, `double`, `bool`, `char`, ...
  - conditionals: `if`, `switch`, ...
  - loops: `for`, `while`
- Basic program structures:
  - Functions
  - Recursive and iterative programming
- Pointers and References
- Classes and Inheritance
  - `class` and `struct`
  - `private`, `public`, `protected`
  - Constructors and Destructors
  - `public`, `private` inheritance
  - (pure) virtual functions abstract base classes
- Polymorphism of functions, operator overloading
- Dynamic memory management (`new`, `delete`)
- Exception handling

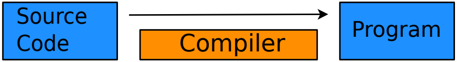
Hello World!

```
// include I/O-library
#include <iostream>

// main is always the first function to be called
// argc: program argument counter
// argv: pointer to C-Strings containing the arguments
int main(int argc, char** argv)
{
  std::cout << "Hello, world..." << std::endl;

  // return value of main
  return 0;
}
```

Establishing the executable necessitates only a compiler (g++):

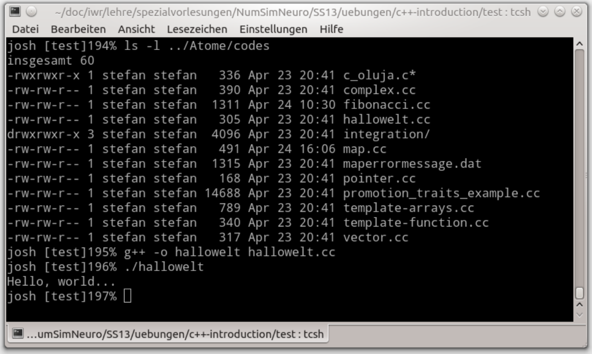


Source Code

Compiler

Program

Compilation with Linux



```
josh [test]194% ls -l ../Atome/codes
insgesamt 60
-rwxrwxr-x 1 stefan stefan   336 Apr 23 20:41 c_oluja.c*
-rw-rw-r-- 1 stefan stefan   390 Apr 23 20:41 complex.cc
-rw-rw-r-- 1 stefan stefan  1311 Apr 24 10:30 fibonacci.cc
-rw-rw-r-- 1 stefan stefan   305 Apr 23 20:41 hallowelt.cc
drwxrwxr-x 3 stefan stefan  4096 Apr 23 20:41 integration/
-rw-rw-r-- 1 stefan stefan   491 Apr 24 16:06 map.cc
-rw-rw-r-- 1 stefan stefan  1315 Apr 23 20:41 maperrormessage.dat
-rw-rw-r-- 1 stefan stefan   168 Apr 23 20:41 pointer.cc
-rw-rw-r-- 1 stefan stefan 14688 Apr 23 20:41 promotion_traits_example.cc
-rw-rw-r-- 1 stefan stefan   789 Apr 23 20:41 template-arrays.cc
-rw-rw-r-- 1 stefan stefan   340 Apr 23 20:41 template-function.cc
-rw-rw-r-- 1 stefan stefan   317 Apr 23 20:41 vector.cc
josh [test]195% g++ -o hallowelt hallowelt.cc
josh [test]196% ./hallowelt
Hello, world...
josh [test]197% 
```

For larger projects the C++-build process is typically quite complicated.

Compilation Process in C++

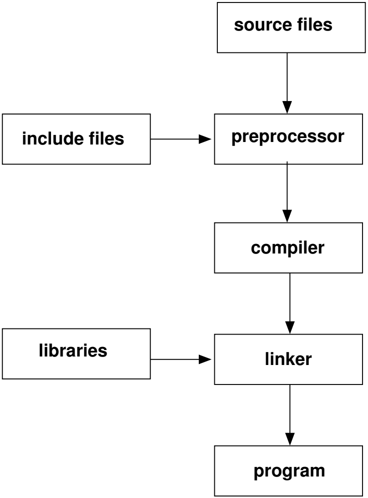
The fine granular construction of an executable program in C++ is coordinated in several steps:

Build Process

- The **preprocessor** analyzises the code and performs substitutions on textual basis (i.e. the substitution of macros and equivalent).
- The **compiler** generates herefrom the **object code**, this means, it analyzeses which objects are necessary and have to be constructed.
- The object code is linked by the **linker** with other libraries and construct the executable program.
- The control of the process is performed via **makefiles**, that are however nowadays mostly hidden in the IDE.

Compilation Process in C++

The following figure shows an overview of the steps to construct an executable program in C++:



Data Types in C++

The elementary data types in C++ are:

int	Integers	`int a = 2;`
long	Large Integers	`long a = 1e15;`
char	Characters	`char a = 'b';`
float	Floating point numbers 4 Byte	`float b = 3.14;`
double	Floating point numbers 8 Byte	`double c = 3.1415;`
bool	boolean values	`bool d = false;`

Branches

`if`-branches:

```
#include <iostream>

int main(int argc, char** argv)
{
  int a = 5;   // an integer variable
  if (a > 0)
  {
    std::cout << "Hello, World..." << std::endl;
  }
  else
  {
    return 1; // emit an error
  }

  return 0;
}
```

Realisation of Loops

- for loops,
- while loops,
- do..while loops.

```
#include <iostream>

int main(int argc, char** argv)
{
  for (int i=1; i<10; ++i)
    std::cout << "i: " << i << std::endl;

  int j = 5;
  while (j > 0)
  {
    std::cout << "j: " << j << std::endl;
    j--;
  }

  return 0;
}
```

Functions

Functions

Functions are needed for encapsulation of program sections and can be called when necessary.

In C++ their syntax always is

```
return-value function-name(parameter1, parameter2, ..);
```

An Example Program with Function

```
#include <iostream>

using namespace std; // use namespace std globally (here ok,
                     // avoid this in the general case)

// A function that greets everyone
void greet()
{
  // do not need namespace-selector std:: any more
  cout << "Hello, World." << endl;
}

// main function
int main(int argc, char** argv)
{
  greet();
  return 0;
}
```

Call-by-Reference und Call-by-Value

In Call-by-Value the address of the object is passed as function parameter and no object copy is constructed:

```
// call-by-value
void swap_wrong (int a, int b)
{
  int tmp = a;
  a = b;        // does not work, a and b are local copies
  b = tmp;      // in the scope of the function
}

// call-by-reference
void swap_right (int& a, int& b)
{
  int tmp = a;  // a, b are reference parameters
  a = b;        // That means changes to them are
  b = tmp;      // persistant after end of function call
}
```

Call-by-Reference und Call-by-Value

```
// main function
int main(int argc, char** argv)
{
  int a=5, b=6;

  // Output 5, 6
  swap_wrong(a, b)
  std::cout << a << ", " << b << std::endl;

  // Output 6, 5
  swap_right(a, b)
  std::cout << a << ", " << b << std::endl;

  return 0;
}
```

Shall changes of a function be persistent always reference variablen have to be used (see in `swap_right`).

Pointer and References

One of the more complicated themes in C/C++ are pointers and references.

Pointer and the address operator &

- `int x = 12`
  The variable x is defined by adress, size (necessary storage demand), name and contents.
- To evaluate the value of the address (not the variable `x`!) the **Adressoperator** & is realized:

```
std::cout << &x << std::endl // Output: 0xA0000000
```

- Address values can be stored in **pointer variables**. Pointer variables have the syntax `Typ* name`, type ist the type of the object, on which the pointer points:

```
int* z = &x; // z is a pointer variable
```

Pointer and References

The dereference operator *

- Using the pointer variable `z`

```
int* z = &x; // z is a pointer variable
```

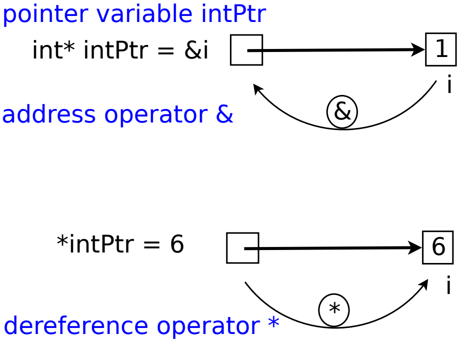
the value of the variable x can also be changed. Herefor exists the (**dereference operator** `*`):

```
*z = 4711; // z is dereferenced, x has now the value 4711
```

- Caution:
  - With the dereference operator the pointer `z` is not changed. (`z` points still onto the memory address of x).
  - The symbol `*` denotes according to the context a dereference operator or a pointer variable.

Pointer and References

The relationhip between pointer variable, adress- and dereference operator is clarified in the following figure:



Pointer and References

References

Besides pointer variables there are *references*.

- References are internal pointers.
- References can be considered as „another name“ for a variable:

```
int   x = 5;
int& y = x;  // another name for x
y = 4;       // means x = 4!
```

Pointer and References

Example for pointer and references:

```
int i, j, *p, *q;
int &s = i, &r = j; // references have to be initialized

r = 2;        // OK, j (==r) has now value 2
r = &j;       // BAD, &j has worng type 'int *' instead of 'int'

p = 2;        // BAD, 2 has wrong type 'int' instead of 'int *'
p = &j;       // OK, p contains now the address of j

if (p == q)   // TRUE, if p, q point to the same address
              // the contents of the address does not matter.

if (r == s)   // TRUE, if the contents of j (reference of r) and i
              // (reference of s) is equal. The adress of the
              // variable does not matter!
```

Pointer and References

(Multi-dimensional) arrays are nothing else than pointer onto the first array entry:

```
int a[5];           // Array of 5 int variables

a[0] = 3;
std::cout << *a;  // output: 3 (= a[0])
std::cout << &a;  // output: adress of a[0]

int a[3][20];       // 3 x 20 array
```

Pointers and References

Pointer enable arbitrary complicated constructs:

```
int **p;          // p contains a pointer onto variables pointing
                  // onto type 'int'

int *p[10];       // p is an array, that contains 10 int * variables,
                  // though the brackets [] bind stronger than *.
                  // this means int * is the type of the array elements!

int (*p)[10];     // Now instead p is a pointer onto an array
                  // with 10 int-components

int* f()          // f is a parameterless function, that
                  // returns a pointer onto an int.
                  // Rounded brackets bind stronger, as above!
```

Memory Segments in C++ programs

In C++ there are esentially thress memory segments where objects can be stored. These are:

Memory segments in C++

1. The *global memory segment*. It stores all global variables and static components of classes and compiled directly into the executable file.
2. The *stack* contains all instances of currently executed methods and functions and their related local variables.
3. The *heap* provides memory, that can be allocated for dynamically allocated objects.

The management of dynamic memory is performed in C++ by the operatoren `new` and `delete`.

Memory Allocation with new

Memory space can allocated from the heap with `new`:

```
int* intPtr;

intPtr = new int; // intPtr points onto the new int memory
```

Withe the code line `intPtr = new int;` memory space is *allocated* for nameless object of type `int` and pointer to it is returned.

The construct `new int`

- reserves space in the heap for an `int` value,
- provides a pointer onto the allocated memory space.

Freeing of Memory with delete

Allocated memory space should be freed when an object is not necessary anymore. This happens with the instruction `delete`:

```
int* intPtr;

intPtr = new int;   // intPtr points onto the new int memory

delete intPtr;      // memory is freed
```

Life Cycle of Objects

Die life time of objects depends on the structure of the program:

- Static and global variables exist during the complete run time.
- Local variables exist as long as the function, they belong to, exist. They are created and destroyes with each new instance.
- Dynamic objects in the heap exist independently of the program structure, their life time is controlled by new and delete.

Life Cycle of Objects

The following code clarifies the different life times of dynamic and static variables:

```
int foo() {
  int* p = new int;      // Generate an nameless variable in the heap
  *p = 5;                // The nameless variable is initialized with 5.
  return p;              // A pointer onto a nameless variable
}                        // is returned. Bad!

void main(void){
  int* q = foo();        // q is generated is initialized with a pointer
  ...                    // onto the nameless variable.
  delete q;              // The nameless variable in the heap is destroyed
  q = NULL;              // OK, q ist now secured (point to nothing)
  ...
}                        // Program end: variable q is deleted
```

Classes and Data Types

A C++ class defines a data type. A data type is a status set with operations, that transform states into each other. Example complex numbers:

```
#include <iostream>

class ComplexNumber { // a class defintion
public:
  void print()
  {
    std::cout << u << " + i * " << v << std::endl;
  }

private:
  double u, v;
};                      // ';' is very important!

int main(int argc, char** argv)
{
  ComplexNumber a, b, c;
  a.print();            // print unitialized (!) number

  //c = a + b; // where defined?

  return 0;
}
```

Classes and Data Types

- C++ enables the encapsulation of a data type, this means separation of implementation and interface.
  - `public`: Interface specification,
  - `private`: Data and implementation.
- From outside only methods and data in the `public` part can be accessed.
- Implementation of methods can happen outside of the class.

Constructors

- The instruction `ComplexNumber a;` make the compiler generate an instance of the class.
- For initialisation the constructor is called.
- There can exist several constructors (polymorphism!).
- In certain cases the compiler generates default constructors.

Constructors

The class `ComplexNumber` with two constructors:

```
class ComplexNumbers
{
public:
  // some constructors
  ComplexNumber() { u = 0; v = 0; }    // default

  ComplexNumber(double re, double im) // initialize with
  { u = re; v = im; }                 // given numbers

  void print() { ... }

private:
  double u, v;
};
```

Constructoren

```
// usage of the complex number class
int main (int argc, char** argv)
{
  ComplexNumber a(3.0,4.0);
  ComplexNumber b(1.0,2.0);
  ComplexNumber c;

  a.print();          // output: 3 + i * 4
  c = a + b;          // where defined ?

  return 0;
};
```

Destructors

- Dynamic generated objects can be destructed, if they are not necessary any more.
- Deletion of objects is handled by the destructor.
- Destructors are especially to be (self)implemented, if the class contains pointer (e.g. arrays!).
- Furthermore when dynamic memory is used inside a class.
- Keywords for dynamic memory management: `new`, `delete`.

Overloading of Operators

Operations for abstract data types (classes)

- The instruction `a + b` is not defined for `ComplexNumber` and must be defined.
- For classes different operations e.g.
  `++,+,*,/,-,--,=,!=,!,==,[],...`
  can be self-implemented.
- Classes, that implement the operator `()` are called *Functors*.

Templates

Templates – Code Patterns

- Templates enable the parameterisation of classes and functors.
- Templates decouple functions or algorithms from data types.
- Allowed parameters:
    - Standard types like `int`, `double`, ...,
    - Own types (classes),
    - Templates.
- Templates enable static polymorphism (see later).
- Templates generalize code → „Generic Programming“.

Example: Templated Function

```
#include <iostream>

// example for a function template
template <class T>
T getMax(const T& a, const T& b)
{
  return (a>b) ? a : b;
}

int main ()
{
  int    i = 5, j = 6, k;
  double l = 10.4, m = 10.25, n;

  k = getMax<int>(i,j); n = getMax<double>(l,m);
  std::cout << k << ", " << n << std::endl;
  // output: 6, 10.4

  return 0;
}
```

Example: Templated Array Class

```
// a class that takes a template parameter
template <typename T> class Array
{
public:
  int add(const T& next, int n);           // add 'next' at data[n]
  T& at(int n);
  T& operator[](int n) { return at(n); } // overloaded operator

private:
  T data[10];
};

// add a new data member
template <class T> int Array<T>::add(const T& next, int n)
{
  if (n>=0 && n<10)
  {
    data[n] = next; return 0;
  }
  else return 1;
}
```

Example: Templated Array Class

```
// get a certain data member
template <class T> T& Array<T>::at(int n)
{
  if (n>=0 && n<10) return data[n];
}

// main program
#include <iostream>
int main()
{
  Array<int> c; c.add(3,0); c.add(4,5); c.add(0,1);
  std::cout << c.at(5) << std::endl;
  // output: 4

  Array<char> d; d.add('x',9);
  std::cout << d.at(9) << std::endl;
  // output: x

  return 0;
}
```

Further on Templates

- Templates are the foundation of generic programming in C++!
- Templates can be self-specialized (for special cases).
- Further template parameter are possible.
- Parameters may have default values.

STL – The Standard Template Library

In C++ ther are many preexisting template container, that can be used for implementation purposes. They are collected in a library, named STL.

The STL

- is a collection of template classes and algorithms,
- provides many container classes (class, that mananges a set of objects),
- has therefore standardized user interfaces for the containers,
- is contained in the C++ standard library.

Container Types of the STL

The STL provides different kinds of containers:

- Sequential container
  Examples: Vectors, lists
- Container adapter
  Restricted Interface for arbitrary containers
  Example: Stacks, queues
- Associative container
  Key-Value Container
  Example: Maps, Multimaps

Dis/Advantages of the STL

Advantages and disadvantages of the STL

- \+ Dynamic memory management
- \+ Avoidance of array overruns
- \+ High quality of containers
- \+ Optimizability by static polymorphism

- – Very complicated, unstructured error messages
- – High demands for compiler and developer
- – Not all compilers are STL-ready (despite the STL is contained in the C++ standard)

Example for the Application of STL containers: vector

```cpp
#include <iostream>
#include <vector>

int main() {
  // example usage of an STL vector
  int result = 0;
  std::vector<int> x(100);

  for (int j=0; j<100; j++) x[j] = j;

  x.push_back(100);

  for (int j=0; j<x.size(); j++)
    result += x[j];

  // output: 5050
  std::cout << result << std::endl;

  return 0;
}
```

The Iterator Interface

Iterators provide access onto the elements of a container. They

- iterate over the elements of a container,
- provide pointer onto container elements,
- are provided by every container class,
- have „r“- and „w“ variants,
- help to avoid array overflows.
- are used in many STL algorithms like sorting, searching and others.

Example: Iterators over a Map

```
#include <iostream>
#include <map>
#include <cstring>

int main()
{
  // example usage of an STL-map
  std::map <std::string, int> y;

  y["one"] = 1; y["two"] = 2;
  y["three"] = 3; y["four"] = 4;

  std::map<std::string, int>::iterator it;
  //std::map¡std::string, double¿::iterator it; // nice error message :-)
  for (it=y.begin(); it!=y.end(); ++it)
    std::cout << it->first << ": " << it->second << std::
          endl;
    // output: one: 1
    // two: 2 ... usw.

  return 0;
}
```

An Disadvantage of the STL: The error messaage

If in this example the wrong type of an iterator is instantiated, the compiler returns the followoing error message:

```
map.cc: In function 'int main()':
map.cc:15: error: no match for 'operator=' in 'it = y.std::map<_Key,
    _Tp, _Compare, _Alloc>::begin [with _Key = std::basic_string<char
    , std::char_traits<char>, std::allocator<char> >, _Tp = int,
    _Compare = std::less<std::basic_string<char, std::char_traits<
    char>, std::allocator<char> > >, _Alloc = std::allocator<std::
    pair<const std::basic_string<char, std::char_traits<char>, std::
    allocator<char> >, int> >]()'
/usr/include/c++/4.4/bits/stl_tree.h:154: note: candidates are: std::
    _Rb_tree_iterator<std::pair<const std::basic_string<char, std::
    char_traits<char>, std::allocator<char> >, double> >\& std::
    _Rb_tree_iterator<std::pair<const std::basic_string<char, std::
    char_traits<char>, std::allocator<char> >, double> >::operator=(
    const std::_Rb_tree_iterator<std::pair<const std::basic_string<
    char, std::char_traits<char>, std::allocator<char> >, double>
    >\&)
map.cc:15: error: no match for 'operator!=' in 'it != y.std::map<_Key,
     _Tp, _Compare, _Alloc>::end [with _Key = std::basic_string<char,
     std::char_traits<char>, std::allocator<char> >, _Tp = int,
    _Compare = std::less<std::basic_string<char, std::char_traits<
    char>, std::allocator<char> > >, _Alloc = std::allocator<std::
    pair<const std::basic_string<char, std::char_traits<char>, std::
    allocator<char> >, int> >]()'
[...]
```

Algorithms

Algorithms provided by the STL

The STL contains many helpful algorithms, that

- manipulate elements of data containers,
- use iterators for element access.

Examples:

- Sorting
- Searching
- Copying
- Reversing the ordering in the container
- . . .

Algorithms

Example: Sorting-Algorithms for Vectors

- Different sorting orders for vectors are usable
- Distinction i.e. by:
  - Used comparison operators
  - Area of sorting
  - Stability
- Complexity of Standard-Sorters for Vectors:
  - $O(n \cdot \log n)$ ideal
  - $O(n^2)$ most unfavourable case
- Self-implemented comparision functions possible
- Caution: (double linked) lits are optimized for insertion and deletion of elements ⇒ special sorting algorithms

Algorithms

Example: Usage of a sorting algorithm for vectors

```
// a vector for integers
vector<int> x;

x.push_back(23); x.push_back(-112);
x.push_back(0); x.push_back(9999);
x.push_back(4); x.push_back(4);

// sort the integer vector
sort(v.begin(), v.end());

// output: -112 0 4 4 23 9999
for (int i = 0; i<x.size(); i++)
  cout << x[i] << "\t";
```

Inheritance in C++

Inheritance

- Data type passes its abstractions to other data types.
- „Is-an“ relation: Triangle is a geometric object, this means is to derive from class GeomObject.
- Not to interchange with a „Contains-a“ relation: A triangle contains three points (but a triangle is no point → no inheritance).

Inheritance in C++

```
// example of inheritance in C++
class Matrix
{
public:
  ...
private:
  double data[3][3]; // (3 x 3)-Matrix
};

// the derived class: symmetrical matrix is a matrix
class SymMatrix: public Matrix
{
public:
  double getEntry(int i, int j) { return data[i][j]; }
        // error: data private in base class
  ...
  // constructor calls a constructor of base class
  SymMatrix() : Matrix() { ... }
};
```

Different Types of Inheritance in C++

In inheritance you have to take care of which members the derived class can access → different types of inheritance:

- `private` inheritance:
  all elements of the base class get private members of the derived class.
- `public` inheritance:
  `public` members of the base class get `public` members of the derived class, `private` gets `private`.

Different Types of Inheritance in C++

- Private member of the base class stay always private (otherwise the encapsulation make no sense).
- Problem: `private` members are encapsulated too strong, `public` members not in anyway.
- Ausweg: `protected` members can access onto derived classes.

Virtual Functions

Virtual Functions enable the hiding of methods of the base class by the derived class:

```
class GeomObject     // base class for geo objects
{                    // 'area' is a function member
public:

  virtual double area() { return 0.0; }
  ...
};

class Triangle : public GeomObject
{                    // a derived class
public:              // has a specific member 'area' as well

  double area()  { return 0.5 * a * h; }
  ...
private:

  double h, a;
};
```

Virtual Functions

When Basis- and derived class members contain the same name – Which method is then called?

```
int main() {
  GeomObject* geo;
  Triangle t;

  geo = &t;
  std::cout << geo->area() << std::endl; // ??

  return 0;
};
```

Solution:

- If specified otherwise the methods of the basis object (!).
- By the keyword `virtual` the call is passed to the derived class.
- Keyphrase **Late Binding**, i.e. mapping method name ⟷ implementation during run time.

Dynamic Polymorphism

The techniqueu of late type-bindung with virtual functions has its own name:

Dynamic Polymorphism

- Exact type determination during runtime.
- Realisation by:
  - Virtual functions (*Function Lookup Table*),
  - Overloading of functions.

Dynamic Polymorphism

The techniqueu of late type-bindung with virtual functions has its own name:

Dynamic Polymorphism

- Exact type determination during runtime.
- Realisation by:
  - Virtual functions (*Function Lookup Table*),
  - Overloading of functions.

Advantages of dynamic polymorphism

- Base class are supersets of derived classes.
- Algorithms, that operate on the base class, can operate on the derived classes as well.
- Example: List, that stores pointers onto `GeomObject`s Pointer can point onto a `Triangle`-Objekt or every other `GeomObject`-Objekt!

Abstract Base Classes and Interfaces

Often virtual functions are not to be define meaningful inside the base class. Then

- Declararation of function in the base clas as „pure virtual“:
- Derived class have to implement pure virtual functions.

Classes with one (or more) pure virtual functions are denoted **abstract base classes**. They are pure interface specifications.

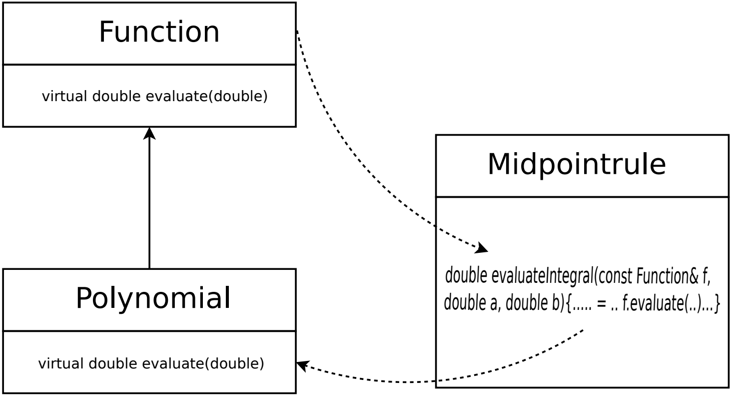
Abstract Base Classes and Interfaces

Abstract Base Classes

- Contains a base class at least on pure virtual function, the class is called abstract.
- From abstract classes no objects can be instanciated.
- An abstract base class defineds an unique interface.
- Algorithms operate on this interface, this means independent of the actual implementation.

Abstract Base Classes and Interfaces

Example:



Function

virtual double evaluate(double)

Midpointrule

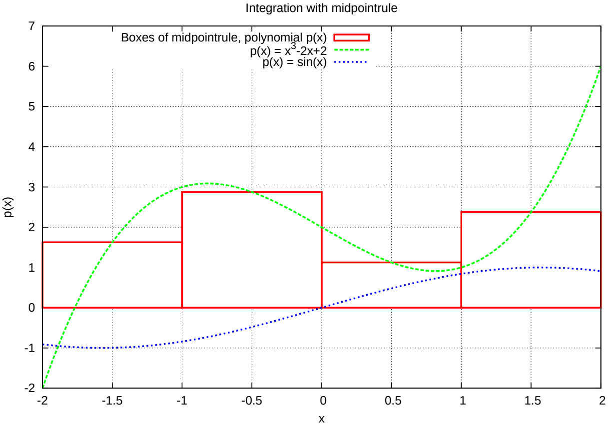
double evaluateIntegral(const Function& f, double a, double b){..... = .. f.evaluate(..)...}

Polynomial

virtual double evaluate(double)

Abstract Base Classes and Interfaces

Example:



Abstract Base Classes and Interfaces

Explanation of the example:

- The algorithm `Midpointrule` integrates arbitrary functions
- It exists an (evtl. abstract) base class for functions
- General functions like polynomials, Sinus, . . . are derived from the base class.
- `Midpointrule` operates only on the functional interface!

It follows the code for the example, a sinus is integrated:

Abstract Base Classes and Interfaces

```cpp
// main.cpp: Test of integration with function interface

// include system header
#include <cstdlib>
#include <iostream>
#include <cmath>

// include own header
#include "sinus.h"
#include "midpointrule.h"

// main function
int main(int argc, char** argv)
{
    // instantiate object of class midpointrule
    MidpointRule mipor(100);

    // generate Sinus object
    Sinus s1;

    // test integration of polynomials
    std::cout << "Integral Sinus: " << mipor.evaluateIntegral(s1,-2.0,2.0) << std::endl;
    std::cout << "Integral Sinus: " << mipor.evaluateIntegral(s1,-3.1415,6.2890) << std::endl;
    std::cout << std::endl;

    return 0;
}
```

Abstract Base Classes and Interfaces

```cpp
// midpointrule.h: The class midpointrule

#include "function.h"

#ifndef __MIPOREGEL_H_
#define __MIPOREGEL_H_

// clase midpointrule
class MidpointRule
{
public:
    MidpointRule(int count) : n(count) {}
    ~MidpointRule() {};

    // evaluate integral of function
    double evaluateIntegral(Function& f, double a, double b) const
    {
      double res = 0.0;
      double h = (b-a)/(1.0*n);          // interval length

      // sum components of individual boxes
      for (int i=0; i<n; ++i)
      {
          double x = a + i*h + 0.5*h; // interval midpoint
          res += h * f.evaluate(x);  // function evaluation
      }

      return res;
    }

private:
    int n;
};

#endif
```

Abstract Base Classes and Interfaces

```
// function.h: Abstract Interface Class for Functions

// Inclusion guards
#ifndef __FUNCTION_H_
#define __FUNCTION_H_

// Abstract base class for functions
class Function
{
public:
  // Constructors
  Function() {};

  // virtual destructor
  virtual ~Function() {};

  // evaluate function, purely virtual !
  virtual double evaluate(double x) const = 0;

private:
};

#endif
```

Abstract Base Classes and Interfaces

```
#include <cmath>

// include base class / interface
#include "funktion.h"

#ifndef __SINUS_H_
#define __SINUS_H_

// encapsulation class for Sinus
class Sinus : public Function
{
public:
    Sinus() {}

    // conform to interface
    double evaluate(double x) const
    {
      return sin(x);
    }

private:
};

#endif
```

Static vs. Dynamic Polymorphism

Dynamic Polymorphism

- The „completely normal“ polymorphism.
- Application: Interface definitions using abstract base classes.
- Enables exchange during runtime.
- Avoids a multiple of optimizations, i.e.
    - inlining,
    - loop unrolling.
- Additional overhead (function lookup tables).

Static vs. Dynamic Polymorphism

Dynamic Polymorphism

- The „completely normal“ polymorphism.
- Application: Interface definitions using abstract base classes.
- Enables exchange during runtime.
- Avoids a multiple of optimizations, i.e.
    - inlining,
    - loop unrolling.
- Additional overhead (function lookup tables).

Static Polymorphism

- Enables exchangeability during compile time only.
- Allows all optimizations.
- Longer compile times.
- Reduces the overhead of the interface.

Static vs. Dynamic Polymorphism

Techniques for Realization of Polymorphisms:

static:

- Templates
- Overloading of functions
- „Engine“ techniques

dynamic:

- Virtual functions
- Overloading of functions

→ Static polymorphism enables to separate algorithms and data structures (interfaces), is evaluated during compile time and enables excessive optimization.

Beispiel: Dynamic polymorphismus in class matrix

```
// base class
class Matrix {
  virtual bool isSymmetricPositiveDefinit();
};

// symmetric matrices
class SymmetricMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit() { ... };
};

// upper triangular matrices
class UpperTriangularMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit()
  { return false };
};
```

The request „Is the matrix symmetric positive definite is passed from the base classe to the derived class.

Beispiel: Dynamic polymorphismus in class matrix

```
// base class
class Matrix {
  virtual bool isSymmetricPositiveDefinit();
};

// symmetric matrices
class SymmetricMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit() { ... };
};

// upper triangular matrices
class UpperTriangularMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit()
  { return false };
};
```

⇒ The approach with virtual functions is in this case eventual not performant. Way out: Static polymorphism (here: engine concept).

The Engine Concept

```
// example delegation of a method to an engine
template<class Engine> class Matrix {
  Engine engineImp;

  bool IsSymmetricPositiveDefinit()
  { return engineImp.isSymPositiveDefinite(); }
};

// some engine classes
class Symmetric {
  bool isSymPositiveDefinite()
  { /* check if matrix is spd. */}
};

class UpperTriangle {
  bool isSymPositiveDefinite(){ return false; }
};
```

The Engine Concept

```
// usage (compiler evaluates Type of A !)
UpperTriangle upper;                    // create upper matrix

Matrix<UpperTriangle> A(upper);        // pass upper to some
                                        // constructor of A

std::cout << A.isSymPositiveDefinite() << std::endl;
```

The Engine Concept

The Engine Approach

- Aspects of different matrices are „packed“ into the engines (`Symmetric` or `UpperTriangular`).
- `Matrix` delegates most of the operations to the engine – during compile time!
- Dynamic polymorphism is substituted by static (templates).
- Disadvantage: The base type (`Matrix`) has to contain all methods of *all* subclasses.
- The trick to avoid this is called „Barton-Nackmann-Trick“.

The Barton-Nackmann-Trick

Also known as *Curiously Recursive Template Pattern*:

```
template<typename LeafType> class Matrix {
public :
  LeafType& engineImp

  void LeafType asLeaf()
  { return static_cast<LeafType&>(*this); }

  bool IsSymmetricPositiveDefinit()
  { return asLeaf().isSymPositiveDefinite(); }
};

// former engine classes derive from base class now!
class Symmetric : public Matrix{
  bool isSymPositiveDefinite()
  { /* check if matrix is spd. */ }
};

class UpperTriangle : public Matrix {
  bool isSymPositiveDefinite(){ return false; }
};
```

The Barton-Nackmann-Trick

```
// usage (compiler evaluates Type of A !)
UpperTriangle upper;      // create upper triangle matrix
Symmetric     sym;        // create symmetric matrix

Matrix<UpperTriangle> A(upper);
Matrix<UpperTriangle> B(sym);

std::cout << A.isSymPositiveDefinite() << std::endl;
std::cout << B.isSymPositiveDefinite() << std::endl;
```

The Barton-Nackmann-Trick

What exactly happenes here during the call `A.isSymPositiveDefinite()`?

- `A` is an object of the base class with template parameter of the derived class.
- Call to `A.isSymmetricPositiveDefinit()` casts `A` onto object of the derived class,
- and calls `isSymmetricPositiveDefinit()` of the derived class!

Motivation

Templates parameterize classes and functions with types. Advanced techniques enable further parameterization:

- Traits – Meta informations of template parameters
- Policies – Behaviour modification of algorithms

Traits

Traits

Represent natural and additional properties of a template parameter.

Examples:

- Meta informations for grids (Is grid conforming, adaptive, . . .)?
- Type promotions.

Type Promotion Traits

Consider addition of 2 vectors:

```
template<typename T>
std::vector<T> add(const std::vector<T>& a,
                   const std::vector<T>& b);
```

Question: Return type when adding two vectors of different types:

```
template<typename T1, typename T2>
std::vector<???> add(const std::vector<T1>& a,
                     const std::vector<T2>& b);
```

Example:

```
std::vector<int> a;
std::vector<complex<double> > b;

std::vector<???> c = add(a, b);
```

Type Promotion Traits

Der Rückgabetyp ist abhängig von den beiden Input-Typen! Das Problem kann mit Promotion-Traits gelöst werden:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1, T2>::promoted_type>
add(const std::vector<T1> &, const std::vector<T2> &);
```

Return type of promotion traits class defines:

```
template<> // promote int to double number
struct Promotion<double, int> {
  public:
    typedef double promoted_type;
};

template<> // promote int to double number
struct Promotion<double, int> {
  public:
    typedef double promoted_type;
};
```

Type Promotion Traits

Example application:

```
std::vector<int>    a(100, 3);
std::vector<double> b(100, 3.1415);

c= add(a, b); // is equivalent to
c= add(b, a); // !
```

Type Promotion Traits

Are many type promotions necessary it simplifies the work of small macros:

```
#define DECLARE_PROMOTE(A,B,C) \
  template<> struct Promotion<A,B> { \
    typedef C promoted_type; \
  }; \
  template<> struct Promotion<B,A> { \
    typedef C promoted_type; \
  };

DECLARE_PROMOTE(int, char, int);
DECLARE_PROMOTE(double, float, double);
DECLARE_PROMOTE(complex<float>, float, complex<float>);
// and so on...

#undef DECLARE_PROMOTE
```

Further Example for Type Promotion

```cpp
#include <iostream>

using namespace std;

// start with the basic template:
template <typename T1, typename T2>
struct Promote
{
};

// the same types are the same
template <typename T1>
struct Promote<T1,T1>
{
    typedef T1 type;
};

// specilizations for all the type promotions
template<> struct Promote<int,char> { typedef int type; };
template<> struct Promote<double,int> { typedef double type;
    };
```

Further Example for Type Promotion

```
// an example function build minima of two variables with different type
template <typename T1, typename T2>
typename Promote<T1,T2>::type min( const T1 & x, const T2 &
     y )
{
     return x < y ? x : y;
}

// main
int main()
{
     std::cout << "min: " << min(88.9, 99) << std::endl;
     // output: 88.9

     std::cout << "min: " << min(4756, 'a') << std::endl;
     // output: 97

     return 0;
}
```

Template Meta Programming

Most important technique of static polymorphism are templates. With templates a programming style for meta programs has evolved:

Template Meta Programs

- Idea: The compiler acts as interpreter.
- Substitute control structures like `if` and loops by specialisation and recursion.
- Theoretical: Turing machine by template programming.

Example of a Template Meta Program: Faculty (T. Veldhuizen)

```
// factorial realized as TMP
template<int N> class Factorial
{
public:
  enum { value = N * Factorial<N-1>::value };
};

// a specialization is needed to break
class Factorial<1>
{
public:
  enum { value = 1 };
};
```

⇒ the value $N!$ is available at compile time as `Factorial<N>::value` by generation of an object of the class:

```
Factorial<12> a; // gives 12!
```

Further Example: Fibonacci Numbers

The following listing demonstrates a program, that evaluates Fibonacci numbers at compile time and run time und measure the times:

```
// fibonacci.cc:
// Compute fibonacci numbers at run- and compile time and compare
// the time used for it.
#include <iostream>
#include <cstdio>

// recursive runtime variant
unsigned long Fibonacci_Simple(unsigned long n)
{
  if      (n==0) return 0;
  else if (n==1) return 1;
  else
    return Fibonacci_Simple(n-1) + Fibonacci_Simple(n-2);
};

// recursive template instantiations
template<unsigned long N>
class Fibonacci
{
public:
  enum { value = Fibonacci<N-1>::value +
                 Fibonacci<N-2>::value };
};
```

Further Example: Fibonacci Numbers

The following listing demonstrates a program, that evaluates Fibonacci numbers at compile time and run time und measure the times:

```cpp
// template specializations to abort iterative template instantiation
template<>
class Fibonacci<1> {
public:
  enum { value = 1 };
};

template<>
class Fibonacci<0> {
public:
  enum { value = 0 };
};

// main program
int main()
{
  // call of recursive Fibonacci
  clock_t begin_rec = clock();
  unsigned long result = Fibonacci_Simple(45);
  clock_t end_rec = clock();
  printf("Recursive Fib(45) = %ld  computed in %lf secs.\n",
          result, (double)(end_rec - begin_rec)/CLOCKS_PER_SEC);
```

Further Example: Fibonacci Numbers

The following listing demonstrates a program, that evaluates Fibonacci numbers at compile time and run time und measure the times:

```

  // call of templated Fibonacci
  begin_rec = clock();
  result = Fibonacci<45>::value;
  end_rec = clock();
  printf("Templated Fib(45) = %ld  computed in %lf secs.\n",
          result, (double)(end_rec - begin_rec)/CLOCKS_PER_SEC);

  return 0;
}
```

Zeiten bei mir für $n = 45$:

- Recursive function: 31 s (since not optimized by cashed values.)
- Templates : 0 s (of course :-)).

Template Meta Programming

Why do we need Template Meta Programs?

- Idea: Hybrid approach, a partitioning of the program in
  - a TMP, executed at compile time
  - a „normal program“

  ⇒ Runtime enhancements (e.g. by massive inlining)
- Generic programming and TMP are used nearly always, if a library simultaneously has to be:
  - with good performance and
  - flexible!
- Specialised algorithms for „small“ classes
- Example: complex numbers, tensors, grids, ...

Template Specialisation

An important technique when using templates is the so called „template specialisation“:

- differences to the template pattern are implemented explicitly,
- I.e. for data types that can be implemented run time and memory efficient.

Template Specialisation

Example for specialisation of templates: Sorting

```
// a sorter class with two template parameters
template <class T, int N> class Sorter
{
  void sort(T* array) { /* sort here */ };
  ...
};

// sorting a single field array is simple...
template <class T> class Sorter<T,1>
{
  void sort(T* array) {};
  ...
};
```

Template Specialisation

Why do we need template specialisation?

Many algorithms (also non-templated ones) can be accelerated by specialisation
Example:

```
// dot-product
double dotproduct(const double* a, const double* b, int N)
{
  double result = 0.0;
  for (int i=0; i<N; i++)
    result += a[i]*b[i];
  return result;
}

// specialization for small N (e.g. N=3) speeds up calculation
double dotproduct(const double* a, const double* b, 3)
{
  return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

Singletons

Singleton Classes

- secure maximal one instantiated object of a class
- substitute global variables
- can orccur in differend designs

Example from Scientific Computing: Quadrature formula for integration, reference elements.

Singletons

Design Example for Singleton Classes

- Class is created inside the class as `static` `member`
- Creation and Return by a function declared `static`
- Constructor is `private`, which avoids multiple instantiations
- Copy constructor is `private` to avoid copy construction by the compiler that otherwise automatically generates copy constructor.

Singletons

Design Example for Singleton Classes

- Class is created inside the class as `static member`
- Creation and Return by a function declared `static`
- Constructor is `private`, which avoids multiple instantiations
- Copy constructor is `private` to avoid copy construction by the compiler that otherwise automatically generates copy constructor.

Singletons

Simplest Realization of a Singleton Class

```
class Singleton
{
public:
  static Singleton& CreateInstance()
  {
    if (mySingleton == NULL)
      mySingleton = new Singleton singleton;
    return singleton;
  }
  static void DeleteInstance()
  {
    if (mySingleton)
      delete mySingleton;
    mySingleton = NULL;
  }
  ... // other public members
```

Singletons

Simplest Realization of a Singleton Class

```
  ... // other public members

private:
  static MySingletonClass* mySingleton;

  Singleton() {};                               // private constructor
  Singleton(const Singleton&);                  // prevent
                                                // copy-construction
  Singleton& operator=(const Singleton&);  // prevent assignment
};
Singleton* Singleton::mySingleton = NULL;

...
// somewhere in the main program:
// create a Singleton instance and get a pointer to it
Singleton* aSingleton = Singleton::CreateInstance();
```

Exceptions

Idea of exception handling in C++:

- Occurs an error somewhere an error flag is set (error throwing).
- An error handling system guards over the error flags (error catching)
- Has an error flag been set the system starts the handling of the exception.

Advantage: An error can be thrown across function limits!

Exceptions

try, throw, catch

In C++ exception handling occurs with `try`-`throw`-`catch` blocks:

- `try` block: Try to execute instruction.
- A `throw` statement in a try block throws an error
- In a `catch` block the thrown error can be catched and an appropriate reaction can be implemented.

Exceptions

```
try{
  ...                        // instructions
  throw exception();         // raise error if it occurs
                             // (Exception is an error class)
  initialise();              // error can be raised in subfunctions
  ...
}
catch(std::exception & e)
{
 ...                         // handle error
}
```

Advanced Literature

There exist many books concerning with proposed optimization possibilites by the presented techniques (especially static polymorphism).

Literature for „Scientific Computing with C++“

- N. Josuttis: C++ Templates – The Complete Guide
- T. Veldhuizen: Techniques for Scientific C++
- T. Veldhuizen: Template Metaprogramming
- E. Unruh: Prime Number Computation (historical example for Template Meta Programming)