Praktikum: Entwicklung interaktiver eingebetteter Systeme

C++-Tutorial

Joachim Falk (`falk@cs.fau.de`)

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

The "++" of C++

- C with additional
  - features of object oriented languages
  - operator and function overloading
  - virtual functions
  - call by reference for functions
  - template functionality
  - exception handling

- Object Orientation is the sum of
  - abstract data types (classes)
  - data hiding
  - (multiple) inheritance
  - polymorphism

Classes - Introduction

```
typedef int t_fifo;
class fifo {
public:
  fifo(int size);
  ~fifo();



  bool read(t_fifo &val);
  bool write(const t_fifo &val);
  void dump();

protected:
  bool is_empty();
  bool is_full();
  int  inc_idx(const int &idx);
protected:
  t_fifo *_buf; int _size;
  int _rd_idx;
  int _wr_idx;
  int _num_elem;
}; // Note the semicolon
```

- A struct in C
  - contains data elements
  - used to encapsulate a state
- A class in C++
  - contains data elements
  - contains functions (called methods)
  - used to encapsulate state and behavior

function members (methods)

data members

Classes - Declaration Syntax

- A Class in C++ is Declared
  - Either using the keyword **struct**
    Which is still there to maintain compatibility with ANSI C
  - Or using the keyword **class**
    Which better fits the object oriented terminology

- Default access modifier (explained later)
  - **public** for **struct**
  - **private** for **class**

```
Syntax:

class class_name {
// implicit private:
  // the class body
}; // Note the semicolon
```

```
Syntax:

struct class_name {
// implicit public:
  // the class body
}; // Note the semicolon
```

Classes - Access Modifier

- Access modifiers define accessibility of class members from outside the class

  - **public** (default for **struct**)
    Members can be accessed from outside the class

  - **protected**
    Members can only be accessed by methods of derived classes

  - **private** (default for **class**)
    members can only be accessed by methods of the class itself

```
class my_class {
  int _value;
public:
  int get_value();
};
```

equivalent

```
struct my_class {
  int get_value();
private:
  int _value;
};
```

Classes - Constructor Syntax

- Every class has a constructor which is a special member function
  - has the name of the class
  - has no return type (not even void)
- The constructor
  - is automatically called at object instantiation
  - is used to initialize class members to a known state
- If no constructor is defined
  - the compiler automatically generates a default constructor
  - which calls the default constructor for all class members

```
class my_class {
public:
  my_class();    // constructor without parameter
  my_class(int); // constructor with int parameter
};

int main(int argc, char *argv[]){
  my_class x;     // calls constructor my_class()
  my_class y(42); // calls constructor my_class(int)
  return 0;
}
```

Classes - Constructor Example

```
typedef int t_fifo;
class fifo {
public:
  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();

protected:
  bool is_empty();
  bool is_full();
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf;
  int     _size;
  int     _rd_idx;
  int     _wr_idx;
  int     _num_elem;
}; // Note the semicolon
```

```
// constructor with int parameter
fifo::fifo(int size) {
  _size = size;
  _buf = new t_fifo[_size];
  _num_elem = 0;
  _wr_idx   = 0;
  _rd_idx   = 0;
  for(int idx = 0;idx < _size;++idx) {
    _buf[idx] = 0;
  }
}
```

```
int main(int argc, char *argv[]){
  // create a fifo of size 32
  fifo y(32);
  return 0;
}
```

Classes - Destructor Syntax

- Every class has a destructor which is a special member function
  - has the name of the class prefix with "~"
  - has no return type (not even void) and no parameters
- The destructor
  - is automatically called right before object destruction
  - is used to cleanup resources used by the class
- If no destructor is defined
  - the compiler automatically generates a default destructor
  - which calls the default constructor for all class members

```
class my_class {
  char *mem;
public:
  my_class(int size); // constructor allocate mem
  ~my_class();         // destructor cleanup mem
};

int main(int argc, char *argv[]){
  my_class y(42); // calls constructor my_class(int)
  return 0;       // before main is left destructor ~my_class is called
}
```

Classes - Destructor Example

```
typedef int t_fifo;
class fifo {
public:
  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();

protected:
  bool is_empty();
  bool is_full();
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf;
  int     _size;
  int     _rd_idx;
  int     _wr_idx;
  int     _num_elem;
}; // Note the semicolon
```

```
// constructor with int parameter
fifo::fifo(int size) {
  _size = size;
  _buf = new t_fifo[_size];
  _num_elem = 0;
  _wr_idx   = 0;
  _rd_idx   = 0;
  for(int idx = 0;idx < _size;++idx) {
    _buf[idx] = 0;
  }
}
```

```
fifo::~fifo() {
  delete[] _buf;
}
```

```
int main(int argc, char *argv[]){
  // create a fifo of size 32
  fifo y(32);
  return 0;
}
```

Classes - Scope Resolution

```
typedef int t_fifo;
class fifo {
public:
  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();

protected:
  bool is_empty();
  bool is_full();
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf;
  int     _size;
  int     _rd_idx;
  int     _wr_idx;
  int     _num_elem;
}; // Note the semicolon
```

The :: operator is called scope resolution operator. It tells the compiler that `read()` and `write()` belong to the class `fifo`.

```
bool fifo::read(t_fifo &val) {
  // do something
}

bool fifo::write(const t_fifo &val) {
  // do something
}
```

C/C++ - Headers and CPPs

```
// Code in header file fifo.hpp
typedef int t_fifo;
class fifo {
public:
  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();
protected:
  bool is_empty()
   { return _num_elem==0; }
  bool is_full()
   { return _num_elem==_size; }
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf;
  int     _size;
  int     _rd_idx;
  int     _wr_idx;
  int     _num_elem;
}; // Note the semicolon
```

```
// Code in implementation fifo.cpp
#include “fifo.hpp”

fifo::fifo(int size) {
   _size=size; _buf=new t_fifo[_size];
   _num_elem=0; _wr_idx=0; _rd_idx=0;
   for(int idx=0; idx<_size; ++idx)
     { _buf[idx] = 0; }
}
fifo::~fifo()
 { delete[] _buf; }

bool fifo::read(t_fifo &val)
 { /* do something */ }
bool fifo::write(const t_fifo &val)
 { /* do something */ }
void fifo::dump()
 { /* do something */ }
int fifo::inc_idx(const int &idx)
 { /* do something */ }
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

References

- C++ supports references to variables
  - a reference to a variable may be generated
    (a reference is an *alias* for the variable)
  - modifying a reference to a variable implies modifying the original variable
  - a reference has to be initialized with the variable which is referenced
    (once initialized the referenced variable cannot be changed)

```
Syntax:

// type_name is the data type of the reference and variable
type_name &ref_name = variable_name;
```

```
int x = 10;
int &y;      // Does not compile as a reference has to be initialized
int &y = x; // This is the correct syntax
y++;         // now y == 11 AND x == 11 (y is just an alias for x)
```

Pointers

- C and C++ supports pointer variables
  - pointers variables contain **memory addresses** as their values.
  - pointers variables can be dereferenced by the *** operator** (called dereferencing operator) which provides the contents in the memory location specified by the pointer variable
  - **memory addresses** for variables may be obtained by the **& operator** (called address operator) which produces the memory address of the variable to which it is applied.

```
Pointer variable declaration syntax:
type_name *ptrvar_name; // for addr. containing data of type type_name
Dereferencing operator syntax:
*ptrvar_name // a value of data type type_name
Address operator syntax:
&variable_name // a address of type "type_name *"
```

```
int x = 10;
int *y; // a pointer variable to a memory location containing an int
y = &x; // now y points to the memory addresses of x
(*y)++; // x == 11 (as *y is an alias for x)
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

Function Argument Default Values

- A function argument may have a default value
  - has to be given in the function prototype (only!)
  - if a default value is given for an argument
    - the argument may be omitted
    - the default value will be used for the argument
  - if a function has more than one argument
    - specification of default values has to start with last argument
    - omission of parameters has to start with the last parameter

```
…
class fifo {
public:
  // constructor now with default argument
  fifo(int size = 16);
  …
};
…
int main(int argc, char *argv[]){
  fifo x;     // create fifo x of default size 16
  fifo y(32); // create fifo y of size 32
  return 0;
}
```

Call by Value

- A function argument which is not a reference is called **“by value”**
  - an argument which is passed “by value” is copied
  - only the copy is modified in the function body
  - at the calling block the passed argument remains unmodified

```
void testfunction(int x, int *z) {
  x++;       // increment x (by one)
  *z += 10; // add 10 to the integer value pointed to by z
  z = &x;   // change z to point to x
  (*z)++;   // increment the integer value pointed to by z, that is x
}

int main(int argc, char *argv[]){
  int  x = 10;
  int  y = 11;
  int *z = &y;
  testfunction(x, z);
  // x is still 10 and z still points to y but y is now 21
  return 0;
}
```

Call by Reference

- A function argument which is a reference is called **"by reference"**
  - does not create a temporary variable for the argument (avoids copying!)
  - if the argument is modified inside the function, the argument variable inside the calling block is also modified
  - often not what you want – use **const reference** instead

Function argument val is passed as a reference to read().

```
bool fifo::read(t_fifo &val) {
  if (is_empty()) {
    return false;
  } else {
    val = _buf[_rd_idx];
    _rd_idx = inc_idx(_rd_idx);
    _num_elem--;
    return true;
  }
}
```

Therefore, the read() method directly modifies y.

```
int main(int argc, char *argv[]){
  fifo x;
  int y = 0;
  x.write(42);
  x.read(y);
  // now y has the value 42
  return 0;
}
```

Call by Reference

- Call by reference may increases program speed
  - no temporary objects created at function calls
    - if passed objects are large this will significantly increase program speed
  - if argument should not be modified by the function
    - use the **const** keyword
    - if the function tries to modify the argument, a compiler error will be issued

Function argument val is passed as a reference to read().

```
bool fifo::read(t_fifo &val) {
  if (is_empty()) {
    return false;
  } else {
    val = _buf[_rd_idx];
    _rd_idx = inc_idx(_rd_idx);
    _num_elem--;
    return true;
  }
}
```

Function argument val is passed as a const reference to write().

```
bool fifo::write(t_fifo const &val) {
  if (is_full()) {
    return false;
  } else {
    _buf[_wr_idx] = val;
    _wr_idx = inc_idx(_wr_idx);
    _num_elem++;
    val = 42; // <= compile error!!!
    return true;
  }
}
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

Function Overloading

- A function may have more than one implementation
  - called overloading
  - the functions must have different type or number of arguments
    - called signature
  - it is not sufficient to have different return types

The two read() functions have a different number of arguments, so overloading is o.k.

```
class fifo {
  …
  bool    read(t_fifo &val);
  t_fifo read();
  …
};
```

Implementing one read() function using the other read() function.

```
t_fifo fifo::read() {
  t_fifo tmp;
  bool success = read(tmp);
  return success ? tmp : -1;
}
```

Function Overloading

- A function may have more than one implementation
  - called overloading
  - the functions must have different type or number of arguments
    - called signature
  - it is not sufficient to have different return types

The two bar() functions are only distinguished via their return types. We get a compiler error!

```
class foo {
  bool   bar(fifo &x);
  double bar(fifo &x);
};
```

Functions can be called without using their return value. So the compiler cannot distinguish between the two functions.

```
int main(int argc, char *argv[]){
  foo  f;
  fifo y(42);
  f.bar(y); // Uh oh which bar?
  return 0;
}
```

Operator Overloading

- Operators are treated as normal functions in C++
  - possible to overload operators
  - operators are usually class members
    - the right operand is passed as argument
    - the left operand is implicitly the class implementing the operator

The operator is declared const, that is the operator cannot modify the class. It can therefore also be used on const objects!

The comparison x == y calls x.operator==(y).

```
class fifo {
public:
  …
  bool operator==(const fifo &rhs) const {
    if(_size != rhs._size)
      return false;
    for(int idx = 0;idx < _size;++idx)
      if (_buf[idx] != rhs._buf[idx])
        return false;
    return true;
  }
  …
};
```

```
fifo x, y;
…
if (x == y)
  …
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

Template Classes - Introduction

- C++ supports a template mechanism
  - allows to specialize classes with parameters
    - especially useful to create classes that can be used with multiple data types
    - extensively used by SystemC
  - the template parameters have to be compile time constants
  - the complete implementation of a template class has to appear in the header (.hpp) file

```
template <class T>
class fifo {
public:
  typedef T t_fifo;
  …
protected:
  t_fifo *_buf;
  …
};
template <int W>
struct bar {
  bar();
  …
  char[W] _char_array;
};
```

_buffer in x is of type “int *”
_buffer in y is of type “float *”

```
int main(int argc, char *argv[]){
  fifo<int>   x;
  fifo<float> y;
  bar<10> a;
  bar<42> b;
  return 0;
}
```

_char_array in a is of size 10
_char_array in b is of size 42

Template Classes - Example

- re-implementing the **fifo** class to be usable with arbitrary data types

```
// Code in header file fifo.hpp
typedef int t_fifo;
class fifo {
public:


  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();
protected:
  bool is_empty()
   { return _num_elem==0; }
  bool is_full()
   { return _num_elem==_size; }
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf; int _size;
  int      _rd_idx;
  int      _wr_idx;
  int      _num_elem;
};
```

```
// Code in implementation fifo.cpp
#include "fifo.hpp"

fifo::fifo(int size) {
  _size=size; _buf=new t_fifo[_size];
  _num_elem=0; _wr_idx=0; _rd_idx=0;
  for(int idx=0; idx<_size; ++idx)
    { _buf[idx] = 0; }
}

fifo::~fifo()
 { delete[] _buf; }

bool fifo::read(t_fifo &val)
 { /* do something */ }

bool fifo::write(const t_fifo &val)
 { /* do something */ }

void fifo::dump()
 { /* do something */ }

int fifo::inc_idx(const int &idx)
 { /* do something */ }
```

Template Classes - Example

- re-implementing the **fifo** class to be usable with arbitrary data types

```cpp
// Code in header file fifo.hpp
template <class T>
class fifo {
public:
  typedef T t_fifo;

  fifo(int size);
  ~fifo();

  bool read(t_fifo& val);
  bool write(const t_fifo& val);
  void dump();
protected:
  bool is_empty()
   { return _num_elem==0; }
  bool is_full()
   { return _num_elem==_size; }
  int  inc_idx(const int& idx);
protected:
  t_fifo *_buf; int _size;
  int     _rd_idx;
  int     _wr_idx;
  int     _num_elem;
};
```

```cpp
// NOW THE IMPLEMETATION NEEDS
// TO ALSO BE IN THE HEADER!!!
template<class T>
fifo<T>::fifo(int size) {
  _size=size; _buf=new t_fifo[_size];
  _num_elem=0; _wr_idx=0; _rd_idx=0;
  for(int idx=0; idx<_size; ++idx)
    { _buf[idx] = 0; }
}
template<class T>
fifo<T>::~fifo()
 { delete[] _buf; }
template<class T>
bool fifo<T>::read(t_fifo &val)
 { /* do something */ }
template<class T>
bool fifo<T>::write(const t_fifo &val)
 { /* do something */ }
template<class T>
void fifo<T>::dump()
 { /* do something */ }
template<class T>
int fifo<T>::inc_idx(const int &idx)
 { /* do something */ }
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

Inheritance - Introduction

- Inheritance enables re-use of components
  - put common features of multiple classes in base class
  - derive classes from the base class
    - all existing features may be re-used
    - new features may be added
  - inheritance establishes a "is-a" relationship
    - e.g., a car is a vehicle, so it could be derived from vehicle
  - access modifiers specify the access to the base class for
    - the derived class
    - derived classes of the derived class
    - everyone else

Syntax:

```
class derived_class : [access_modifier] base_class {
  // class body
};
```

Inheritance - Access Modifier

Overview of access modifiers for inheritance

base class \ derived as	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	no access	no access	no access

```
Class base {
public:
  void pub_func();
protected:
  void prot_func();
private:
  void priv_func();
};
```

```
class cls1: public base {
  // pub_func()  is still public
  // prot_func() is still protected
  // priv_func() cannot be accessed from cls1
};
class cls2: protected base {
  // pub_func()  is now   protected
  // prot_func() is still protected
  // priv_func() cannot be accessed from cls2
};
class cls3: private base {
  // pub_func()  is now private
  // prot_func() is now private
  // priv_func() cannot be accessed from cls3
};
```

Inheritance - Example

```
// code in resizebale_fifo.hpp
typedef int t_fifo;

class resizeable_fifo: public fifo {
public:
  resizeable_fifo(int size = 16);
  ~resizeable_fifo();
  void resize(int size);
};
```

```
// code in main.cpp
int main(int argc, char *argv[]){
  // create a resizeable fifo
  // of size 32
  resizeable_fifo x(32);
  // resize fifo to 42 elements
  x.resize(42);
  // write data to the fifo
  x.write(10);
  return 0;
}
```

The constructor of `resizeable_fifo` calls the constructor of its base class with the size argument.

```
// code in resizebale_fifo.cpp
resizeable_fifo::resizeable_fifo(int size): fifo(size)
{}
void resizeable_fifo::resize(int size) {
  // a resize destroys all stored data
  delete[] _buf;
  _size = size;
  _buf = new t_fifo[size];
}
```

Multiple Inheritance - Introduction

- Multiple Inheritance
  - derive a class from multiple base classes
  - extensively used by SystemC
    - necessary to allow separation of interface and implementation of a channel
  - multiple inheritance is an advanced feature of C++
    - only mentioned in this introduction, not covered in depth

`sc_fifo<>` is derived from three base classes

```
template <class T> class sc_fifo : public sc_fifo_in_if<T>,
                                   public sc_fifo_out_if<T>,
                                   public sc_prim_channel
{
…
};
```

Agenda

- Classes
- Pointers and References
- Functions and Methods
- Function and Operator Overloading
- Template Classes
- Inheritance
- Virtual Methods

Virtual Methods - Declaration

- Virtual Methods in C++
  - provide a mechanism to re-implement methods of a base class
    - a method declared virtual *may* be re-implemented within a derived class
  - a so-called pure virtual method
    - *must* be re-implemented in the derived class
    - no implementation provided in the base class
    - enables the implementation of interfaces without any functionality

a virtual method

a pure virtual method without implementation

```
class foo {
public:
  virtual void virt_func() {
    std::cout << "I am a method of foo" << std::endl;
  }
};

class bar { // similar to a java interfaces
public:
  virtual void pure_virt_func() = 0;
};
```

Virtual Methods – Reimplementation

```
class derived_foo: public foo {
public:
  virtual void virt_func() {
    std::cout << "I am a method of derived_foo" << std::endl;
  }
};

class derived_bar: public bar {
public:
  virtual void pure_virt_func() {
    std::cout << "I am not pure virtual any more" << std::endl;
  }
};
```

```
foo x;
x.virt_func();
// bar cannot be instantiated, because it has a pure virtual method
derived_foo y;
y.virt_func();
derived_bar z;
z.pure_virt_func();
```

Output:

```
I am a method of foo
I am a method of derived_foo
I am not pure virtual any more
```