Agenda

C++ Introduction

- 1. Classes
- 2. Functions and Operators
- 3. Template Classes
- 4. Inheritance
- 5. Virtual Functions

esign

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS



raunhofor

Fraunhofer Institut

Integrierte Schaltungen

Fraunhofer Institut Integrierte Schaltungen

The "++" of C++

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

The sum of

- abstract data types (classes)
- data hiding
- (multiple) inheritance
- polymorphism

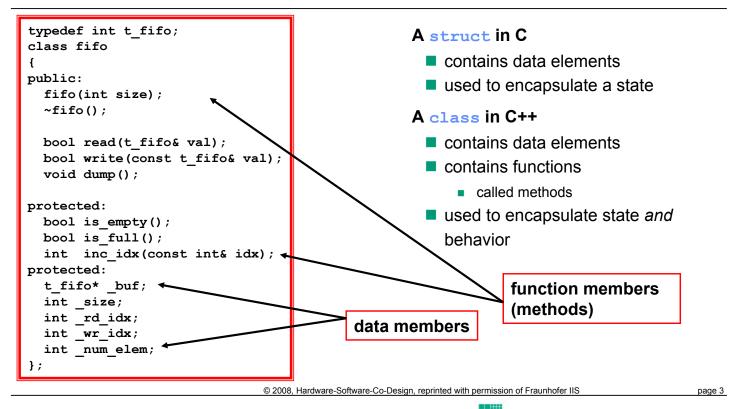
© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

2000



Fraunhofer Institut Integrierte Schaltungen

Classes - Introduction



Classes - Declaration Syntax

A Class in C++ is Declared

Either using the keyword struct

still there to maintain compatibility with ANSI C

Or using the keyword class

better fits the object oriented terminology

The Difference Between

class and struct

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
};

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

Sign Fraunhofer



Classes - Access Modifier

Access Modifiers

Accessibility of class members from outside the class

are available in three different types

public

Members can be accessed from outside the class

default for struct

protected

Members can only be accessed by methods of derived classes

private

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

members can only be accessed by methods of the class itself

equivalent

© 2008. Hardware-Software-Co-Design, reprinted with

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



Fraunhofer Institut

Institut
Integrierte Schaltungen

Classes - Constructor Syntax

Every class has a constructor

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

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



© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page 6



Fraunhofer Institut
Integrierte Schaltungen

The :: operator is called scope resolution operator. It

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;
```

```
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;
    }
}</pre>
```

```
int main()
{
   // create a fifo of size 32
   fifo y(32);
   return 0;
}
```

Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

nage 7

esign



Classes - Scope Resolution Operator

```
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;
```

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
}

dware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

pa





Agenda

2. C++ Introduction

- 1. Classes
- 2. Functions and Operators
- 3. Template Classes
- 4. Inheritance
- 5. Virtual Functions

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS







Functions & Ops. - 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

```
Reference Syntax:

type_name &ref_name = variable_name;

type_name:

the data type of the reference
```

```
int x = 10;
// int &y; FAILURE
int &y = x;
y++; // now y == 11 AND x == 11 (y is just a reference to x)
```

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page 11





Functions & Ops. - Default Value

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()
{
    // create a fifo of default size 16
    fifo x;
    // create a fifo of size 32
    fifo y(32);
    return 0;
};
```



Fraunhofer Institut
Integrierte Schaltungen

Functions & Ops. - Call by Reference 1

C++ supports call by reference for functions

- passing a reference as argument to a function
 - 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

```
bool fifo::read(t fifo &val)
                                    pass a reference as
                                    argument to the function
                                                                   the read() method directly
  if( is empty() ) {
                                                                   modifies v
    return false;
                                            int main()
                                              fifo x;
    val = buf[ rd idx];
                                              int y = 0;
    rd idx = inc idx( rd idx);
                                              x.write(42);
    num elem--;
                                              x.read(y);
                                               // now y has the value 42
  return true;
                                      B, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS
```

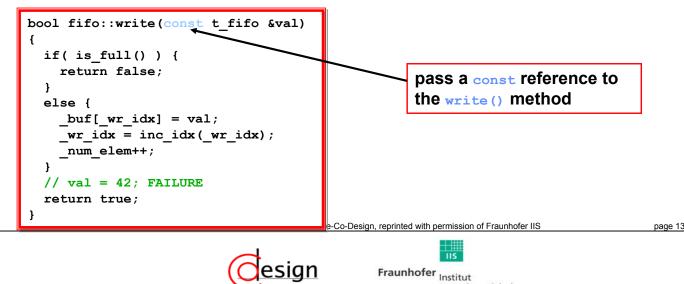


Fraunhofer Institut Integrierte Schaltungen

Functions & Ops. - Call by Reference 2

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



Functions & Operators - Overloading 2

Integrierte Schaltungen

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

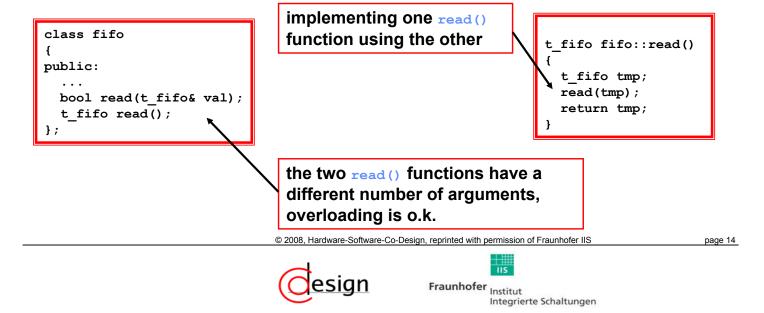
```
class fifo
                                                       The operator is declared const, i.e. the
{ public:
                                                       operator cannot modify the class. It can
                                                       therefore also be used on const objects!
  bool operator==(const fifo& rhs) const;
                                    bool fifo::operator == (const fifo& rhs) const
};
                                    { if( size != rhs. size)
                                        return false;
                                      bool result = true;
                                      for(int idx = 0;idx < size;++idx) {</pre>
                                        result = result && ( buf[idx] == rhs. buf[idx]);
fifo x, y;
                                      return result;}
if (x == y) \dots // calls x.operator==(y)
                                                  are-Co-Design, reprinted with permission of Fraunhofer IIS
```



Functions & Operators - Overloading 1

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



Agenda

2. C++ Introduction

- 1. Classes
- 2. Functions and Operators
- 3. Template Classes
- 4. Inheritance
- 5. Virtual Functions

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page





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 (.h) file

```
buffer in x is of type int*
template <class T> class foo
                                                                    buffer in y is of type float*
  foo();
                                            int main()
private:
                                              foo<int> x;
 T* buffer;
                                              foo<float> y;
template <int W> struct bar
                                              bar<10> a;
                                              bar<42> b;
  bar();
private:
  char[W] _char_array;
                                    © 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS
                                      esign
                                                           Fraunhofer Institut
                                                                    Integrierte Schaltungen
```

be usable with arbitrary data types fifo(int size = 16); ~fifo(); bool read(T& val); code in fifo.h template<class T> bool write(const T& val); inline void dump(); bool fifo<T>::write(const T& val) protected: if(is full()) { bool is empty(); return false; bool is full(); int inc idx(const int& idx); else { _buf[_wr_idx] = val;

code in fifo.h

Template Classes - Example

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS



num elem++;

return true;

Fraunhofer Institut Integrierte Schaltungen

wr idx = inc idx(wr idx);

■ re-implementing the fifo class to

Agenda

2. C++ Introduction

- 1. Classes
- 2. Functions and Operators
- 3. Template Classes
- 4. Inheritance
- 5. Virtual Functions

Inheritance - Introduction

Inheritance enables re-use of components

- put common features of multiple classes in base class
- derive classes from the base class

template<class T> class fifo

public:

protected:

T* buf;

int size;

int rd idx;

int wr idx;

int num elem;

- 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

```
Inheritance Syntax:
class derived class : [access modifier] base class
   access modifier:
    one of the three modifiers public, protected, private
```

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

Fraunhofer Institut

Integrierte Schaltunger





Inheritance - Access Modifier

Overview of access modifiers for inheritance

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

```
class foo
{
  public:
    void pub_func();
  protected:
    void prot_func();
  private:
    void priv_func();
};
```

```
class bar : public foo
{
    // pub_func() is still public
    // prot_func() is still protected
    // priv_func() cannot be accessed from bar
};
```

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page 21





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

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page 2





Inheritance - Example

```
code in main program
                        code in header file
                                                 int main()
                                                 { // create a resizeable fifo of size 32
typedef int t fifo;
                                                   resizeable fifo x(32);
                                                   // resize fifo to 42 elements
class resizeable_fifo : public fifo
                                                   x.resize(42);
                                                   // write data to the fifo
  resizeable fifo(int size = 16);
  ~resizeable fifo();
                                                   x.write(10);
 void resize(int size);
                                                   return 0;
  the constructor of resizeable fifo calls the constructor of its base class with the size
  argument
 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];
                                                      code in implementation
                                 © 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS
                                                       Fraunhofer Institut
```

Agenda

2. C++ Introduction

- 1. Classes
- 2. Functions and Operators
- 3. Template Classes
- 4. Inheritance
- 5. Virtual Functions

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

page 2





Integrierte Schaltungen

Virtual Functions - Declaration

Virtual Functions 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 function
 - must be re-implemented in the derived class
 - no implementation provided in the base class
 - enables the implementation of interfaces without any functionality

```
class foo
{ public:
    virtual void virt_func() {
       cout << "I am a function of foo" << endl;
    }
};
class bar // similar to java interfaces
{ public:
    virtual void pure_virt_func() = 0;
};</pre>
a virtual function
without implementation
```

© 2008, Hardware-Software-Co-Design, reprinted with permission of Fraunhofer IIS

esign



Virtual Functions - Re-Implementation

```
class derived_foo : public foo
{
  public:
    virtual void virt_func() {
      cout << "I am a function of derived_foo" << endl;
    };
  class derived_bar : public bar
  {
  public:
    virtual void pure_virt_func() {
      cout << "I am not pure virtual any more" << endl;
    }
};</pre>
```

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

Output:
I am a function of foo
I am a function of derived_foo
I am not pure virtual any more
© 2008, Hardware-Software-co-pesign, reprinted with permission or reaumnoier its
```

page 26



Fraunhofer Institut Integrierte Schaltungen