# Agenda

## C++ Introduction

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

# 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

# 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
```

**function members (methods)**

**data members**

**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

# Classes| Java equivalent

```cpp
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
```

```java
// typedef int t_fifo not supported in java
public class fifo {

  public         fifo(Integer size) {…}
  public    void finalize(){…}

  public class IntegerRef
   { public Integer val; };

  public    Boolean read(IntegerRef ref)  {…}
  public    Boolean write(Integer val)    {…}
  public    void    dump() {…}

  protected Boolean is_empty() {…}
  protected Boolean is_full()  {…}
  protected Integer inc_idx(Integer idx) {…}

  protected Vector<Integer> _buf;
  protected Integer         _rd_idx;
  protected Integer         _wr_idx;
  protected Integer         _num_elem;
}
```

# 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
};
```

# 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`

**members can only be accessed by methods of the class itself**

- default for `class`

```
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**

- 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);
};
```

```
int main(){
  my_class x;      // calls constructor my_class()
  my_class y(42);  // calls constructor my_class(int)
}
```

# 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;
  }
}
```

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

# 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;
};
```

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

# 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;
};
```

**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++ - Header and Implementation File

```
// 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 file 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

## 2. C++ Introduction

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

# 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;
}
```

# 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)
```

**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)
{
  if( is_empty() ) {
    return false;
  }
  else {
    val = _buf[_rd_idx];
    _rd_idx = inc_idx(_rd_idx);
    _num_elem--;
  }
  return true;
}
```

**pass a reference as argument to the function**

```
int main()
{
  fifo x;
  int y = 0;
  x.write(42);
  x.read(y);
  // now y has the value 42
}
```

**the `read()` method directly modifies y**

# 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

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

**pass a `const` reference to the `write()` method**

**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

```
class fifo
{
public:
  ...
  bool read(t_fifo& val);
  t_fifo read();
};
```

**implementing one `read()` function using the other**

```
t_fifo fifo::read()
{
  t_fifo tmp;
  read(tmp);
  return tmp;
}
```

**the two `read()` functions have a different number of arguments, overloading is o.k.**

## 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
{ public:
  ...
  bool operator==(const fifo& rhs) const;
  ...
};
```

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

```
bool fifo::operator==(const fifo& rhs) const
{ if(_size != rhs._size)
    return false;

  bool result = true;
  for(int idx = 0;idx < _size;++idx) {
    result = result && (_buf[idx] == rhs._buf[idx]);
  }
  return result;}
```

```
fifo x, y;
…
if (x == y) … // calls x.operator==(y)
```

# Agenda

## 2. C++ Introduction

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

# 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

```
template <class T> class foo
{
  foo();
  ...
private:
  T* _buffer;
};
template <int W> struct bar
{
  bar();
  ...
private:
  char[W] _char_array;
};
```

```
int main()
{
  foo<int> x;
  foo<float> y;

  bar<10> a;
  bar<42> b;
  ...
}
```

`_buffer` in `x` is of type `int*`
`_buffer` in `y` is of type `float*`




Fraunhofer Institut Integrierte Schaltungen

# Template Classes - Example

code in fifo.h

```
template<class T> class fifo
{
public:
  fifo(int size = 16);
  ~fifo();

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

protected:
  bool is_empty();
  bool is_full();
  int  inc_idx(const int& idx);

protected:
  T* _buf;
  int _size;
  int _rd_idx;
  int _wr_idx;
  int _num_elem;
};
```

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

code in fifo.h

```
template<class T>
inline
bool fifo<T>::write(const T& val)
{
  if( is_full() ) {
    return false;
  }
  else {
    _buf[_wr_idx] = val;
    _wr_idx = inc_idx(_wr_idx);
    _num_elem++;
  }
  return true;
}
```

# 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
  - 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`

# 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
};
```

# Inheritance - Example

code in header file

```
typedef int t_fifo;

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

code in main program

```
int main()
{ // 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**

```
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 file

 

# 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

## 2. C++ Introduction

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

# 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() {
    std::cout << "I am a function of foo" << std::endl;
  }
};
class bar { // similar to java interfaces
public:
  virtual void pure_virt_func() = 0;
};
```

**a virtual function**

**a pure virtual function without implementation**

# Virtual Functions - Re-Implementation

```
class derived_foo : public foo {
public:
  virtual void virt_func() {
    std::cout << "I am a function 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 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
```