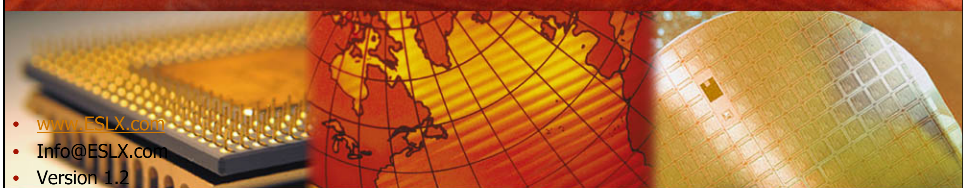
Basic C++ for SystemC

David C Black



- www.ESLX.com
- Info@ESLX.com
- Version 1.2

XtemeESL Corporation – ESL Specialists

- Founded 2003
  - Broad Background (Hardware/Software/Methodology/Systems)
  - Active in SystemC Standardization working groups
  - Authors of book SystemC: From the Ground Up
  - Merged with XtremeEDA Corporation as a US subsidiary July 2008
- Services
  - ESL Adoption Planning
  - Methodology and Flow Definition & Development
    - General Modeling and Software Development Platforms
    - Architectural and Functional Verification
    - Behavioral Synthesis
  - Staffing
    - Mentoring
    - Peak staffing needs
  - Training and Quick Ramp Mentoring
- Clients include small “startups” to Fortune 500

Call us today and let our experts help your company become successful with ESL

Objectives - C++ for SystemC

- Provide a quick C++ review
  - Assumes a knowledge of C
- Make it easier to learn SystemC
  - Focus on elements used by SystemC
- NOT a ground up tutorial
  - See references for that
  - Use as a guideline on what to learn

Fasten your seatbelts!

Agenda - C++ for SystemC

- Nature of C++
- Strings
- Streaming I/O
- Namespaces
- Functions
  - Defining & using
  - Pass by value & reference
  - Const arguments
  - Overloading
  - Operators as functions
- Templates
  - Defining
  - Using
- Classes (OO)
  - Data & Methods
  - Constructors
  - Destructors
  - Inheritance
  - Polymorphism
  - Constant members
  - Static members
  - Guidelines
- STD Library tidbits

History of C++

- In 1980, Bjarne Stroustrup, from Bell labs, began the development of the C++ language, that would receive formally this name at the end of 1983, when its first manual was going to be published. In October 1985, the first commercial release of the language appeared as well as the first edition of the book "The C++ Programming Language" by Bjarne Stroustrup.
- During the 80s the C++ language was being refined until it became a language with its own personality. All that with very few losses of compatibility with the code with C, and wothout resigning to its most important characteristics. In fact, the ANSI standard for the C language published in 1989 took good part of the contributions of C++ to structured programming.
- From 1990 on, ANSI committee X3J16 began the development of a specific standard for C++. In the period elapsed until the publication of the standard in 1998, C++ lived a great expansion in its use and today is the preferred language to develop professional applications on all platforms.

XtremeEDA

Multi-paradigm language

- Procedural programming - C
  - Simple data, Conditionals, Loops & Functions
- Modular programming
  - Namespaces, Exception handling
- Data abstraction
  - Structures, User defined types (enums & simple classes)
  - Concrete types & abstract types
- Object Oriented
  - Class hierarchies, inheritance, overriding, polymorphism
- Generic Programming
  - Templates, Containers, Algorithms

XtremeEDA

C-style “strings”

```
char* msg = “Hello there”;
char  mesg2[80];
```

Dangerous
Avoid

- Really just pointer to unchecked array
  - Danger, Will Robinson! Danger!

```
typedef char* cstr;
cstr name = “K&R”; // Array of 4 chars
#include <cstring>
strcpy(cstr,cstr), strcat(cstr,cstr),
strcmp(cstr,cstr), strlen(cstr), strchr(cstr,c)
#include <ctype.h>
isalpha(c), isupper(c), isdigit(c), isspace(c),
isalnum(c), toupper(c), tolower(c)
```


std:string

```
#include <string>
std::string mesg3(“Hello”);
std::string mesg4;
```

- Much better/safer than C-strings
  - Assign **operator=** and Concatenate **operator+**
    - Dynamically resizes
  - s.**length**()<**string::npos**, s.**size**(), s.**capacity**(), s.**resize**(N), s[*pos*]
  - Compare with operators **==**, **!=**, **>**, **<**, **<=**, **>=**
  - Methods .**insert**(), .**find**(), .**replace**(), .**substr**(), **swap**()

C-style I/O

```
printf(char* fmt,var1, var2, …);
```

- Terse format limited to predefined types
  - "%d %s %f %x %c"
- Not type checked at compile-time

- Guidelines
  - Discouraged in C++ (see next slides)

Streaming I/O

```
#include <iostream>
```

- Streaming I/O makes it more natural

```
cout << "Heading: " << obj << endl;
```

- Objects "output themselves in an appropriate format."
  - No need to remember the correct %d %f %s
  - All output is consistent

Streaming I/O guidelines

- Define for every datatype
  - `ostream& operator<<(const ostream& s, const Data& d)`
  - `{`
    `os << “Data field: ” << d.data … ;`
  - `} // no endl please`
- Also
  - `ofstream& operator<<(const ofstream&, const Data&);`
- Use **boost::format** to aid

C Scope

pie.cpp

```
float joe(3.14159);

~~extern float joe;~~
```

main.cpp

```
void func() {
  signed joe;
  for (long joe = 0; joe!=3; ++joe)
    cout << joe << ‘ ’ << ::joe << endl;
}
int main() {
  char joe = ‘c’;
  { BLOCK:
    double joe = 6.28318; // Hides main joe
    cout << joe << ‘ ’ << ::joe << endl;
    func();
  }
}
```

```
6.28318 3.14159
0 3.14159
1 3.14159
2 3.14159
```

Namespaces - powerful

some.cpp

```
float joe(3.14159); // global
namespace gi { complex joe(2007,1984); }
namespace your { string joe("your"); }
namespace my { namespace gi { short joe(42); }}
```

other.cpp

```
#include "some.h" // externs to above
using your;
namespace my {
  string joe("along");
  void moe() {
    long joe = 96;
    { NESTED:
      char joe = 'c'; // Hides long joe
      cout << ::joe << ' ' << joe << ' ' << gi::joe << ' '
           << ::gi::joe << ' ' << my::joe << endl;
    }
  }
}
int main() { my::moe(); }
```

```
3.14159 c 42 2007+1984j along
```


Namespaces - anonymous

- Good for hiding
- Preferred alternative to file **`static`**

```
namespace {
  int magic = 42;
}
void use_magic() {
  cout << magic << endl;
}
```

Namespace - Guidelines

- Some EDA vendors have restrictions
  - Cadence disallows **`sc_module`** inside **`namespace`**
- Use, but don't abuse
  - Good for modular programming
  - Keeps nests < 2 deep
  - XtremeEDA uses top-level `::XEDA` for library
- Use anonymous instead of **static**
  - For file scoped variables
- Use `::global` for clarity
  - Identifies globals & discourages their use
- Convenience of using
  - Do **`using`** `::SPACE::MEMBER;` as needed
  - Don't **`using namespace`** `SPACE;` in headers

Agenda - Functions

- C++ supports procedural programming
- Functions are the basis for procedures
- The following topics will be covered:
  - Declaring, defining and using functions
  - Passing arguments by value
  - Pass arguments by reference
  - Const arguments
  - Overloading function names
  - Operators as functions

Declaring functions

- Simple indicates the syntax for usage and makes it available for use
- Often included in header (.h) files
- May be repeated without causing errors

```
int main(int argc, char* argv[]);

void display(string message);

float sum(vector v);

void status(void);
```


Defining functions

- Defines behavior
- May only be done once

```
void display(string message) {
  cout << message << endl;
}
typedef vector<int>::iterator vi_t;
int sum(vector<int> v) {
  int total = 0;
  for (vi_t e=v.beg();e!=v.end();++e) {
    total+=*e;
  }
  return total;
}
```

Using functions

- Straight forward

```
display("Hello there");
int y = sum(v) + 3;
status();
```

- It is possible to pass address of function
  - Use in lookup tables
  - As a parameter to a generic algorithm

Passing arguments by value

- Copy supplied arguments into variables
  - Only way in C
- Example

```
void f(int a) {
  a = a+1;
  cout << "a=" << a << endl;
}
int main() {
  int x = 42;
  f(x);
  f(5);
  cout << "x=" << x << endl;
  return 0;
}
```

```
a=43
a=10
x=42
```

Passing arguments by reference

- Make variables point to original argument
  - Had to use messy pointers in C
- Example

New C++ syntax

```
void f(int& a) {
  a = a+1;
  cout << "a=" << a << endl;
}
int main() {
  int x = 42;
  f(x);
  // f(5); ILLEGAL - Cannot modify "5"
  cout << "x=" << x << endl;
  return 0;
}
```

```
a=43
x=43
```


Const arguments

```
int sum(const std::vector<int>& v);
```

New C++ syntax

- Compiler enforces "read-only" use
- Similar to task input in Verilog
- Good for passing large values by reference
- Documents intent

Overloading function names

- Use **same name** for several different functions
  - Distinguished by number of arguments -or-
  - Distinguished by types of arguments
  - This is illegal in C

```
int add(const std::vector<int>& v);
float add(const std::vector<float>& v);
int  add(int* a, int size);
int  add(int a, int b);
int  add(complex a, int b);
int  add(int b, complex a);
void add(float a, complex a, complex& result);
```

- Return type not considered as part of signature

Operators as functions

- a+b is another way of saying add(a,b)
- C++ allows you to overload operators
  - May only use existing operators
  - May not change # arguments or precedence
  - May not redefine existing combinations
    - E.g. may not redefine **int** + **int** (this is goodness)
  - Some operators require reference or const
- Example
  - `complex operator+(complex lhs, complex rhs);`
- Use only where it makes intuitive sense
  - What does car + car = mean?

Topics - Templates

- C++ supports generic programming
- The following topics will be covered:
  - Using
  - Defining
  - Guidelines

Why generic programming?

- Suppose you want to create a struct/class that can hold several data types and perform operations on them cleanly.
  - Could use union, but code has to store information about which data type is currently active, and code has to be duplicated to do different tasks.

Templates (generic programming)

- Using templates is fairly easy & powerful
  - Standard template library (STL) is based on (drum roll) . . . Templates!
  - SystemC uses templates a lot
- Defining templates is a bit messy
  - Guideline: Design a class without templates before you add the details of templatization
- Functions and classes may be templated
  - Most folks familiar with class templates

Ex: using templates

- From STL
  - `list<pixel>       image_list;`
  - `map<string,bool> used;`
- From SystemC
  - `sc_int<12>       reg;`
  - `sc_fifo<int>     int_fifo;`
  - `sc_fifo<packet> pkt_fifo;`
  - `sc_fixed<8,4>    scale;`

Defining templates

- To define a template class use the **template** reserved word and include argument specifications in angle brackets (<>) as shown here

```
template<class T, int N>
struct fifo {
  T buff[N];
  void push(T v);
  T pop();
};

int main() {
  fifo<string,5> s_fifo;
  fifo<int,32> i_fifo;
  s_fifo.push("hello");
  i_fifo.push(50);
}
```


Templates are powerful

- Plate tectonics are powerful too!
- Several types of templates
  - Template classes

    `template<typename T> CLASSNAME {…};`
  - Template functions

    `template<typename T> RETURN FUNC(ARGS) {…};`
- Get basic class working before templating
- Can have several arguments
  - Both typename's and integral values
  - Latter arguments may have defaults

Defining templates can hurt

- An entire book devoted to the subject!
- Must consider disambiguation
  - C++ rules can be challenging
  - Will two classes/functions suffice?
- Quite a few idiosyncrasies
  - Best to use `template<typename T>`
  - `#include "CLASSNAME.cpp"`
  - Use `this->` for members
- Partial & complete specialization

Agenda - Object-Oriented C++

- C++ supports the Object-Oriented (OO) paradigm
- The following topics will be covered:
  - Defining a class
  - Methods
  - Access types
  - Constructors & initialization
  - Destructors
  - Inheritance
  - Initializing base classes
  - Adding members
  - Overriding methods
  - Multiple inheritance
  - Protection & friends
  - Virtual methods
  - Pure virtual
  - Abstract classes
  - Interface classes
  - Virtual inheritance
  - Constant members
  - Static Members

Properties of objects

- Objects are data types
  - Examples: integer; processor; complex number; creature; shape; window
- Objects have state
  - Examples: integer value; processor register contents; real & imaginary portion of complex; size & orientation of a shape; window name, type & color
- Objects have behavior
  - Examples: integer can be added, subtracted, multiplied; processor can execute instructions; complex can be added, multiplied (scalar & cross); shape can be drawn, inquired of size; window can be moved, resized, drawn, closed

What is a class?

- Classes are custom data types
  - Effectively extend a programming language
- Classes define object types
  - Define data such as properties, and state
  - Define behaviors and capabilities
- Classes have:
  - State (member data)
  - Methods (member functions)

Member Data & Member Functions

- In C++, all data types are classes
- Instances of a data type are called objects
  - `int a; // creating an object instance`
- Objects have functions they can perform
  - `a = 5;     // store`
  - `a = a + 5; // retrieve, add & store`
- C++ uses keywords **`struct`** or **`class`**
  - Functions are allowed as members

What is a class in C++?

- In C++, a class is simply a **`struct`** that has at least one member function (aka method).

```
struct NAME {
  void METHOD(); // makes NAME a class
};
```

- By default, all members of a **`struct`** or public (i.e. accessible directly from the outside using the “dot” operator.)
- The keyword **`class`** was introduced to help document intent and almost synonymous to **`struct`** except for a minor detail of access that will be discussed later.

```
class NAME2 { // also a class
  void METHOD(); // makes NAME a class
};
```

Class suggestions

- Comments in the implementation (cpp) should be limited to internal how or why things are done (i.e. implementation notes)
- Separate data from functions
  - Strict OO programming dictates all access to an object should be through member functions.
  - Considered taboo to modify member data of a class
- C++ convention
  - Prefix member data variables with m_.
- Put plenty of usage comments in the header
  - The header is the file that users will see

Creating a class

- Separate specification (declaration) from implementation (definition)
  - Use header file (.h) to specify
  - Use implementation file (.cpp) to define
- Use **`struct`** or **`class`**
  - OO purists prefer **`class`**
  - SystemC historically used **`struct`**, but changed its tune during the standardization process

Ex: tail_light.h (specify a class)

```
#ifndef TAIL_LIGHT_H
#define TAIL_LIGHT_H
class tail_light {
 public: // Member functions - behavior
  bool  is_on();
  void  set_on();
  void  set_off();
  void  set_rate(float duty_cycle); // 0.0 to 1.0
  void  set_rate(bool light[10]);
  float get_rate();
  // Member data - internal state
  bool  m_on;
  bool  m_light[10]; // 1/10th of duty cycle status
};
#endif
```


Using a class

- Treat a class as a new data type (like int)
  - Happens to be user-defined
  - Has unique behaviors
  - `CLASSNAME IDENTIFIER();`
- Use of the member functions follows the same syntax we use with member data in a struct
  - Uses the dot operator
  - `OBJECT.FUNCTION(ARGS…)`

Ex: main.cpp (use a class)

```
#include "headlight.h"
#include "tail_light.h"
int main() {
  // create objects (instantiate)
  headlight  left_front, right_front;
  tail_light left_rear,  right_rear;
  // call member functions
  left.set_rate(0.5);
  left_rear.set_on();
  right_rear.set_off();
  if (left_rear.is_on()) {
    cout << "Left tail light is on" << endl;
  }
}
```


Implementing a class

- Implementation means defining the behaviors of member functions (methods)
- Place implementation in separate .cpp file
- Include the header file
- Use of a namescope operator (`::`) to identify methods (member functions)
  - Indicates function belongs to the class
  - `TYPE CLASSNAME::METHODNAME(ARGS){BODY}`

Ex: tail_light.cpp (1 of 3)

```
#include "tail_light.h"
// Define methods in tail_light
bool tail_light::is_on() {
 return m_on;
}
void tail_light::set_on() {
  m_on = true;
}
void tail_light::set_off() {
 m_on = false;
}
```


Ex: tail_light.cpp (2 of 3)

```
void tail_light::set_rate(float duty) {
  if (duty < 0.0 || 1.0 < duty) {
    cout << "ERROR: Illegal rate "
         << duty << endl;
  } else {
    for (int i=0;i!=10;++i) {
      m_light[i] = (i >= 10*duty);
    }//endfor
  }//endif
}
```

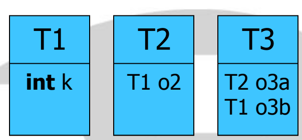
Ex: tail_light.cpp (3 of 3)

```
float tail_light::get_rate() {
  float rate = 0;
  for (int i=0;i!=10;++i) {
    if (m_light[i]) {
      rate += 0.1;
    }//endif
  }//endfor
  return rate;
}
```


The "has a" relationship

- Data members of a class are objects
  - Hierarchies of class instantiations are a powerful way of creating complex classes
  - This is known as **composition**
  - This establishes a **"has a"** relationship
  - For instance:

```
struct T1 { int k; };
struct T2 { T1 o2; };
struct T3 { T2 o3a; T1 o3b};
```



UML class diagrams

  - Class T1 has a **int**
  - Class T2 has a T1
  - Class T3 has a T1 and has a T2

Inline methods

- In the preceding, class declaration (header) was kept separate from class implementation (cpp)
- It is possible to do both in one step

```
struct A {
  int m_v;
  void print() { cout << "v="<<v<<endl;}
};
```

- The method `print` is created **inline** with the code where it is invoke (if possible).
  - Creates very fast code - good
  - Larger executable - ok
  - Exposes implementation to end user
- Use only for extremely simple methods
  - get & set methods are good examples

Ex: inline

```
#ifndef TAIL_LIGHT_H
#define TAIL_LIGHT_H
struct tail_light {
  // Member functions - behavior
  bool  is_on()   { return m_on; }
  void  set_on()  { m_on = true; }
  void  set_off() { m_on = false; }
  void  set_rate(float duty_cycle); // 0.0 to 1.0
  void  set_rate(bool light[10]);
  float get_rate();
  // Member data - internal state
  bool  m_on;
  bool  m_light[10]; // 1/10th of duty cycle status
};
#endif /* TAIL_LIGHT_H */
```

Class Accessibility

- By default all members of a **struct** are public
- It is desirable to hide parts of class from users (e.g. member data or private functions)
- Three keyword labels control access to members of a class:
  - **public:** // anyone can access
  - **private:** // only for members of this class
  - **protected:** // available to family members
    - More on this later
- By default
  - **struct** is public
  - **class** is private

Adding access to a struct

- **struct** default is public
- Public members on right
  - func(), help(), m_y
- Private member on right
  - sub() m_x, m
- T2 is not very useful
  - Cannot acces m!

```
struct T1 {
  int func(float rate);
  void help();
 private:
  int sub(char c);
  int m_x;
 public:
  int m_y;
};
struct T2 {
 private:
  int m;
};
```

Adding access to a class

- `class` default is public
- Public members on right
  - `display(), m_y, m`
- Private member on right
  - `task(), help(), sub() m_x`
- T4 acts like a `struct`

```
class T3 {
  void task(int& w);
  void help();
 private:
  int sub(char c);
  int m_x;
 public:
  void display();
  int m_y;
};
class T4 {
 public:
  int m;
};
```


Ex: Public & Private

```
#ifndef TAIL_LIGHT_H
#define TAIL_LIGHT_H
class tail_light {
public: // Member functions - behavior
  bool  is_on()    { return m_on; }
  void  set_on()   { m_on = true; }
  void  set_off()  { m_on = false; }
  void  set_rate(float duty_cycle); // 0.0 to 1.0
  void  set_rate(bool light[10]);
  float get_rate();
private: // Member data - internal state
  bool  m_on;
  bool  m_light[10]; // 1/10th of duty cycle status
};
#endif /* TAIL_LIGHT_H */
```

Notes on Using Public and Private

- Always precede members of a class with access designations (i.e. public, private)
- When defining classes, prefer the keyword class
- Place public stuff first, private last
  - It's what the user wants to know
- Minimize private stuff

Constructors

- Our tail_light class is missing something
  - Initial values of the member data are unknown
  - Need initialize
- Functional programming suggests adding a member method called reset or initialize
  - Problematic
    - Requires user call every time object is created
    - Experience shows the user will eventually forget
    - Failure to initialize variables difficult to debug

Initialization - the wrong way

- Perhaps we can just initialize?

```
struct tail_light {
  bool  m_on(true);
  bool  m_light[10]= {false,
                       /*etc*/, false };
};
```

- Problematic
  - C++ doesn't allow this syntax
  - m_on(true) syntactically looks like a function defn

Solution: Use a constructor

- C++ has a special syntax for initialization
  - Special method called a **constructor**
- A constructor is a member function that has the <u>same name as the class name</u>, and <u>returns no value</u>:

```
struct CLASSNAME {
  CLASSNAME(ARGS…);
};
```

- Constructor with no args is "default constructor"

Ex: Default Constructor

```
struct tail_light {
  …
  // default constructor
  tail_light(void);
  …
};
```

No return type | Class name | No arguments

```
tail_light::tail_light(void) {
  m_on = true;
  // Default 50% duty cycle
  for (int i=0;i!=10;++i) {
    m_light[i] = (I<5);
  }//endfor
}
```


Constructors with arguments

- Possible to have a constructor take an argument
  - Useful to establish a tail_light with a different initial duty cycle
- Because constructors are simply functions
  - Can overload them the same way as any function
  - Might have both a default constructor (50% duty cycle), and the constructor that takes an argument. A constructor is always invoked when objects are instantiated.

Ex: Constructor (non-default)

```
struct tail_light {
  …
  // constructor with args
  tail_light(int percent);
  …
};
```

No return type

```
tail_light::tail_light(int pct) {
  m_on = true;
  int div = int(10*pct/100+0.5);
  for (int i=0;i!=10;++i) {
    m_light[i] = (i<div);
  }//endfor
}
```


Default constructors

- If you do not provide a constructor, then the "default constructor" is provided for you.
  - Default constructor simply allocates space for the data members (i.e. no initial values).
- If you specify a constructor with one or more arguments, then the "default constructor" will not be provided unless you provide it (i.e. overload).
- If you do not specify a constructor when instantiating, then the "default constructor" is invoked for you.
- If you do not specify a constructor when instantiating and there is no default constructor, then it is an error.

Choosing the constructor

- There is still a potential problem with our approach to initialization
- Consider a class that instantiates a class

```
struct Complex {
  double re; double im;
  // No default constructor
  Complex(double r, double i);
};
struct Amplifier {
  Complex x;
};
```


Initializer lists

- A syntactical construct was added to C++ to allow choosing the constructor for data members

```
CLASSNAME::CLASSNAME(ARGS…)
: ELT(ARGS),… // initializer list
{
  // BODY
};
```

Initializer notes

- Occurs before the body of the constructor is executed
- Using empty parentheses invokes the default constructor for a class
  - For **int**, this means set to zero
- Proceeds in the order data members are declared
  - HINT: List them in the same order as declared
  - If order dependences exist, document them
- Initialization arguments may be an expression
  - valid at construction time

```
class T1 {
  float m_k;
  T1(float k): m_k(k) {
    m_k++;
  }
};
class T2 {
  int m_n;
  T1  m_a1;
  T2() : m_n(), T1(1) {}
};
class T3 {
  int x, y, z;
  T3(): y(1), x(y+1), z(y) {}
};
```


Ex: Initializer list

```
tail_light::tail_light(int pct)
: m_on(true)
{
  int div = int(10*pct/100+0.5);
  for (int i=0; i!=10; ++i) {
    m_light[i] = (i<div);
  }//endfor
}
```

What is a destructor?

- Objects/data are destroyed when
  - Code leaves a scope
  - **delete** is explicitly called
  - program terminates
- It is desirable to do cleanup
  - Free storage
  - Output statistics
  - Delete embedded linked list (avoid leaks)
- For this C++ provides a destructor

Defining the destructor

- C++ destructor is a method named after the class with a preceding tilde (**~**) that takes no arguments (ever) and returns no value (where would it go)

```
CLASSNAME::~CLASSNAME() {
  BODY
}
```

- If you don't provide a destructor, the compiler will provide a default that simply frees member data memory.

Ex: Destructor

- Declaration

```
struct tail_light {
  …
  // destructor
  ~tail_light();
};
```

- Implementation

```
tail_light::~tail_light() {
  cout<<"destroyed tail_light"<<endl;
}
```


Destructor notes

- Called for every object as it is destroyed
- There is only **one** destructor per class
- If you rely on the default destructor, put a comment to that effect in the header.

Inheritance motivation

- Some classes share common attributes
  - Sedan & hatchback automobiles could be modeled as classes
    - Both have 4 wheels, engine, steering, etc.
  - Managers & engineers could be classes
    - Both have names, ages, etc.
  - Circles & squares
    - Both have sizes, positions & orientations
- Desirable to only write code once for common features
- Ability of one class to "inherit" from another
  - Sedan & hatchback inherit from car class
  - Manager & engineer inherit from employee class
  - Circle & square inherit from shape class

How to inherit

- Design the base (parent) class carefully
- Specify the class to inherit with the syntax

```
class DERIVED_CLASS
: PARENT_CLASS_LIST {
 …
};
```

- Parent class list
  - Comma separated
  - Name of class
  - Optional access specifier
  - Syntax

```
public|private|protected CLASSNAME,…
```

Ex: Parent class - light.h

```
#ifndef LIGHT_H
#define LIGHT_H
#include <string>
class light {
public:
  enum Color {WHITE,RED,YELLOW,GREEN };
  light(Color c); // constructor
  light(std::string k, Color c); // constructor
  bool is_on()   {return m_on;   }
  void set_on()  {m_on = true;   }
  void set_off() {m_on = false; }
private:
  Color       m_color;
  bool        m_on;
  std::string m_kind;
};
#endif
```

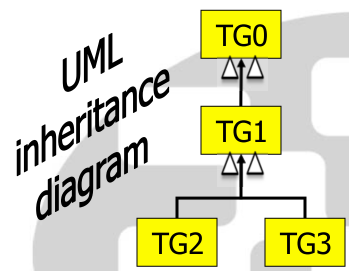

Ex: Inheritance

```
#ifndef TAIL_LIGHT_H
#define TAIL_LIGHT_H
#include "light.h"
class tail_light : public light {
 public: // Member functions - behavior
  tail_light(); // default constructor
  tail_light(int percent_on); // constructor
  ~tail_light(); // destructor
  void  set_rate(float duty_cycle); // 0.0 to 1.0
  void  set_rate(bool light[10]);
  float get_rate();
 private: // Member data - internal state
  bool  m_light[10]; // 1/10th of duty cycle status
};
#endif
```

The "is a" relationship

- A parent class (base) & a child class (derived) use the **"is a"** relationship
  - The child class **"is a"** parent class
  - The converse is not true
- TG1 is a TG0
- TG2 & TG3 are a TG1

```
class TG0 {…}
class TG1:TG0 {…}
class TG2:TG1 {…}
class TG3:TG1 {…}
```



Initialization of inherited classes

- When constructing a class that instantiates another class within it
  - Base (parent) classes are constructed first
- What if you need to specify arguments to base class constructor
  - e.g. parent class has no default constructor
- Use the initializer list!

Ex: Initializer list

```
tail_light::tail_light(int pct)
: light(Red)
{
  int div = int(10*pct/100+0.5);
  for (int i=0;i!=10;++i) {
    m_light[i] = (i<div);
  }//endfor
}
```


Adding Members

- Inheriting class (child or derived class) may define new behaviors and data
  – Sports car has spoiler
  – Manager has ability to approve raises
  – Square has sides
- Simply add new member functions/data

Overriding Inherited traits

- Derived classes may have different data/behaviors for given function
  - Sports car has 2 doors instead of 4
  - Manager attends more meetings
  - Circle draws differently
- Defining the same method again in the derived class effectively hides the parent method

Ex: Overriding methods

```
class tail_light : public light {
 public:
  bool is_on(); // override
  …
};

bool tail_light::is_on() {
  // on if m_on is true and current
  // light cycle is true
  …
}
```

Accessing parent methods

- A derived class can access all the public/ protected members of the base class
  - Even if it overrides the parent

```
BASECLASS::METHOD(ARGS…)
```

- This allows modification of base behavior

```
DERIVEDCLASS::METHOD(ARGS) {
  //pre modifications
  BASECLASS:METHOD(ARGS);
  // post modifications
  return RESULT;
}
```


Multiple inheritance

- C++ allows inheritance from more than one parent class
  - Known as multiple inheritance
  - Used judiciously, it is powerful and useful
- What happens if two base classes have the some common method signatures?
  - Simply override and specify which one rules…
- What if two base classes share a common ancestor (famous diamond problem)?

Protected members

- **`private`** access specification means private to the class where used
  - Children may not access parent's private info
- What about "family" secrets?
  - Use the designation **`protected`**
  - Protected information is available to class where declared and any derived class
- When designing a class must think ahead

Ex: Protected

```
class light {
  public:
   enum Color {WHITE,RED,YELLOW,GREEN};
   light(Color c); // constructor
   bool is_on()    {return m_on;   }
   void set_on()   {m_on = true;   }
   void set_off()  {m_on = false;  }
  protected:
   bool m_on;
  private:
   Color m_color;
};
```

Friends

- What if we would like to extend access to another function or class that is not a part of the family?
  - Specify the function or class as a friend
  - WARNING: Friends can access everything

```
class B;
class A {
  friend B;
};
```

  - Use sparingly

What is polymorphism?

- The ability to have a function or method that takes derived objects as base class arguments and behaves correctly with respect to overridden behaviors.

Why polymorphism?

- Consider a class of shapes
  - A shape might have an inherent ability to draw itself; however…
  - A circle has a unique draw method
    - i.e. overrides base shape::draw
  - A square has a different draw method
    - i.e. overrides base shape::draw
  - It would be nice to be able to have a list of shapes and then just draw each one
- Consider a base printer class
  - Both laser and inkjets have the ability to print
  - Print works differently in the laser and inkjet printers
  - A test function might take a generic printer as a parameter and attempt to print regardless of the sub-class of printer

Ex: Without polymorphism

```
class printer {
 public:
  void print(string s)
  { cerr<<"Base:Oops!"<<endl;
  }
};
class laser : public printer {
 public:
 void print(string s)
  {cout<<"Laser:"<<s<<endl;}
};
class inkjet : public printer
  {
 public:
 void print(string s)
  {cout<<"Inkjet:"<<s<<endl;}
};
```

```
void f(printer p) {
  p.print("hello");
}

int main() {
  printer generic;
  laser lj5550;
  inkjet dj2800;
  f(generic);
  f(lj5550);
  f(dj2800);
}
```

```
% test_print
Base:Oops!
Base:Oops!
Base:Oops!
```

Virtual methods

- To enable polymorphism C++ designates the shared methods as virtual
  - **virtual** RTN_TYPE METHOD(ARGS);
- This causes C++ to create a lookup table in the class, which allows a derived class to specify an overridden function.

Ex: With polymorphism

```
class printer {
 public:
  virtual void print(string s)
  {cerr<<"Base:Oops!"<<endl;}
};
class laser : public printer {
 public:
  void print(string s)
  {cout<<"Laser:"<<s<<endl;}
};
class inkjet : public printer
  {
 public:
  void print(string s)
  {cout<<"Inkjet:"<<s<<endl;}
};
```

```
void f(printer p) {
  p.print("hello");
}

int main() {
  printer generic;
  laser lj5550;
  inkjet dj2800;
  f(generic);
  f(lj5550);
  f(dj2800);
}
```

```
% test_print
Base:Oops!
Laser:hello
Inkjet:hello
```

Pure virtual methods

- It would be nice if we could ensure that all printers had a print function at compile-time instead of a run-time error
- Declaring a method to be pure enables this
  - **virtual** RTN_TYPE METHOD(ARGS)**=0**;
- Think of **=0** as meaning "This function has no implementation."

Abstract & Interface classes

- A class containing a pure virtual method is called an **abstract class**.
- An abstract class cannot be instantiated because there is no definition for the pure virtual method.
- A class containing **only** pure virtual methods (no data either), is call an **interface class**.
- An interface class is effectively an API (Application Programming Interface) for a class.

Ex: With pure virtual

```
class printer {
 public:
  virtual void print(string s)
  =0;
};
class laser : public printer {
 public:
  void print(string s)
  {cout<<"Laser:"<<s<<endl;}
};
class inkjet : public printer
  {
 public:
  void print(string s)
  {cout<<"Inkjet:"<<s<<endl;}
};
```

```
void f(printer p) {
  p.print("hello");
}

int main() {
  //printer generic; ILLEGAL
  laser lj5550;
  inkjet dj2800;
  f(lj5550);
  f(dj2800);
}
```

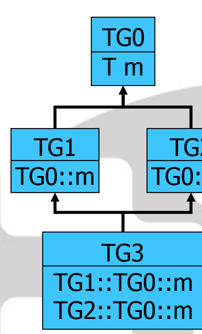
```
% test_print
Laser:hello
Inkjet:hello
```


XtremeEDA

The dreaded diamond

- When inheriting from multiple classes that inherit from a base class, it is possible that duplication of data occurs.
- TG1 & TG2 each have a copy of TG0::m
- TG3 has two copies
  - TG1::TG0::m
  - TG2::TG0::m

```
class TG0 {T m;}
class TG1:TG0 {}
class TG2:TG0 {}
class TG3:TG1,TG2 {}
```

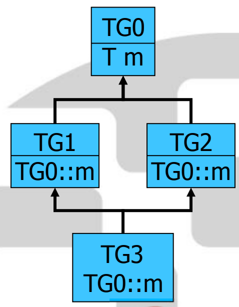


XtremeEDA

Virtual inheritance (Avoiding the dreaded diamond)

- To prevent this, declare the inherited class as **`virtual`**.
- NOTE: This is a completely different concept from virtual methods.

```
class TG0 {T m}
class TG1
: virtual TG0 {…}
class TG2
: virtual TG0 {…}
class TG3:TG1,TG2 {…}
```



Constant members

- Adding the keyword **`const`** to a method restricts the method from modifying any member data

```
class T1 {
  public:
  int get() const { return m; }
  void get(int& v) const { v = m; }
  void set(int v);
 private:
  int m;
};
```

- May not call non-**`const`** methods inside a **`const`**
- Use **`const`** whenever possible
  - Good for get methods

Ex: const members - light.h

```
#ifndef LIGHT_H
#define LIGHT_H
#include <string>
class light {
public:
  enum Color {WHITE,RED,YELLOW,GREEN };
  light(Color c); // constructor
  light(std::string k, Color c); // constructor
  bool is_on() const {return m_on;  }
  void set_on()  {m_on = true;   }
  void set_off() {m_on = false; }
private:
  Color       m_color;
  bool        m_on;
  std::string m_kind;
};
#endif
```


Static members

- Inside ordinary functions, static is used to create variables that have infinite lifetimes. The same is true for classes.
- Static member functions may not alter non-static member data nor call non-static methods.
- Must initialize static member data externally
- Use static to gather statistics for all the objects of an entire class

```
class T1 {
  T1():m(0) {++cnt;}
  ~T1() {--cnt;}
  void set(int v) {m=v;}
  static void count(){
    cout<<k<<endl;}
  int m;
  static int cnt;
};
static int T1::cnt(0);
```

Disabling default methods

- Use **private** or **protected** to disable
  - Sometimes you want to prevent copying or construction (e.g. interfaces)
  - Use comment to clarify intent

```
class no_copy {
  protected: // Disable the following

  no_copy(); // Constructor

  private: // Disable the following for everyone

  no_copy& operator=(const no_copy& rhs) {}
  no_copy(const no_copy& old) {}
};
```


Take control of the class

- Always define and comment constructor(s)
  `CLASSNAME(ARGS…); // Constructor`
- Avoid implicit conversions by using **explicit**
  `explicit CLASSNAME(ARG);`
- Always define or disable the copy constructor & **operator=**
  - At minimum provide a comment // Default copy
  `CLASSNAME(const CLASSNAME&);`
  `CLASSNAME& operator=(const CLASSNAME);`
- Interface classes define API
  - Pure virtual methods have no implementation
  `virtual RETURN METHOD(ARGS) = 0;`
- Destructors are your friend - destroy data leaks
  - Allows correct polymorphism
  `virtual ~CLASSNAME();`

Exceptions

- C++ provides a mechanism to handle exceptions
  - Divide by zero
  - System call errors (e.g. read error)
  - User-defined exceptions ("FIFO underflow")
- SystemC does not currently use exceptions
  - Proposed extensions for modeling do use exceptions
  - Modeling situations may use exceptions

Exceptions in 3 parts

- Easy syntax/concept

class to hold information on the exception

```
class my_exception {
  string msg; my_exception(string m):msg(m){}
};

void some_func():my_exception {
 if (bad_situation) throw my_exception("Oops");
}
```

Function might throw the exception

Throw it

```
try {
some_func();
}
catch (my_exception& problem) {
  REPORT_ERROR(problem.msg);
  if (unrecoverable) throw; //upward again
}
catch (other_exception& problem) {…}
```

try it

Catch it

Exceptions - Caveats

- Always catch by reference
- May confuse threading, so use with care
  - Always catch if thrown unless desire abort
  - Don't expect kernal to understand
  - SC_REPORT_ERROR or SC_REPORT_FATAL may be better for many instances
- Can lead to spaghetti code
  - How much preventative coding do you do?
  - Clean design of classes is important
- Can lead to memory leaks
  - Watch those automatic variables

Safe Code Techniques

- Pass by Value or Reference when possible
  - Less error prone to use by reference than pointers

```
void Func1(long *v_ptr) {
  *v_ptr = 55;
}
long v;
Func2(&v);
```

```
void Func2(long &v) {
  v = 55;
}
long v;
Func2(v);
```

- Use const where possible
  - Avoids possibility of side effects catching you unaware

```
char const * const RCSID = "$Id$";
class myclass {
  double const m_maxval;
  myclass(const double maxval) :m_maxval(maxval) {…}
  bool legal(const double ref&) const;
};
```

Hiding data in a class

- Data hiding provides implementation freedom
- Good for IP (eslx library)

my.h

```
class my_private; // no need to #include header!
class my {
  my(); // Constructor
  virtual ~my(); // Destructor
  private:
  my_private* m;
};
```

Forward declaration

my.cpp

```
struct my_private { // no need for private
  int hidden_int;
  my_private() {…} // constructor
  void hidden_func() {…}
};
my::my(): m(new my_private) {…}
// use m->hidden_int or m->hidden_func()
```

Needs only space for private pointer

To hide or not to hide

- Hiding speeds up compliation
  - No need to parse headers
- May hide too much
  - If need to debug (waveforms), should expose specific data or provide methods to do so.
- SYSTEMC GUIDELINES
  - Ports are public
  - Signals that may need tracing are public

NIH - Use it!

- Standard Template & BOOST Libraries
  - Free, reviewed, debugged
- Quick Overview
  - History
  - **cstring** vs **std::strings**
  - Streaming I/O + **boost::format**
  - **vector**<T>::at(), **list**<T>
  - **map**<T1,T2>, **set**<T>
  - **boost::regex**
  - **boost_shared_ptr**

STL General Background (Wikipedia)

- <http://www.sgi.com/tech/stl/>
- The C++ Standard Library is based on the STL published by SGI. Both include some features not found in the other. SGI's STL rigidly specifies a set of headers, while ISO C++ does not specify header content.
- The architecture of STL is largely the creation of one person, Alexander Stepanov. In 1979 he began working out his initial ideas of generic programming and exploring their potential for revolutionizing software development. Although Dave Musser had developed and advocated some aspects of generic programming as early as 1971, it was limited to a rather specialized area of software development (computer algebra).
- Stepanov recognized the full potential for generic programming and persuaded his then-colleagues at General Electric Research and Development (including, primarily, Dave Musser and Deepak Kapur) that generic programming should be pursued as a comprehensive basis for software development.

Boost General Background

- <http://www.boost.org>
- Free peer-reviewed portable C++ source libraries.
- Emphasizes libraries that work well with the C++ Standard Library and intended to be widely useful, and usable across a broad spectrum of applications.
- Boost license encourages both commercial & non-commercial use. Not GNU.
- 10 Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) as a step toward becoming part of a future C ++ Standard. More Boost libraries are proposed for TR2.
- Why "boost"? Beman Dawes stated "Boost began with Robert Klarer and I fantasizing about a new library effort over dinner at a C++ committee meeting in Sofia Antipolis, France, in 1998. Robert mentioned that Herb Sutter was working on a spoof proposal for a new language named Booze, which was supposed to be better than Java. Somehow that kicked off the idea of "Boost" as a name. We'd probably had a couple of glasses of good French wine at that point. It was just a working name, but no one ever came up with a replacement."

XtremeEDA

Boost List of Functionality - sampler

- **any** - Safe, generic container for single values of different value types, from Kevlin Henney.
- **array** - STL compliant container wrapper for arrays of constant size, from Nicolai Josuttis.
- **assign** - Filling containers with constant or generated data has never been easier, from Thorsten Ottosen.
- **format** - Type-safe 'printf-like' format operations, from Samuel Krempp.
- **math** - Several contributions in the domain of mathematics, includes atanh, sinc, and sinhc
- **numeric/conversion** - Optimized Policy-based Numeric Conversions, from Fernando Cacciola.
- **interval** - Extends the usual arithmetic functions to mathematical intervals
- **multi_array** - Multidimensional containers and adaptors for arrays of contiguous data, from Ron Garcia.

XtremeEDA

Boost List of Functionality - sampler

- **random** - A complete system for random number generation, from Jens Maurer.
- **rational** - A rational number class, from Paul Moore.
- **regex** - Regular expression library, from John Maddock
- **uBLAS** - Basic linear algebra for dense, packed and sparse matrices, from Joerg Walter and Mathias Koch.
- **smart_ptr** - Five smart pointer class templates, from Greg Colvin, Beman Dawes, Peter Dimov, and Darin Adler.
- There are many others…

STL Containers

- Vectors, the better array

```
#include <vector>
std::vector<float> fv(50,0.0);
for(int I=0; I!=fv.size(); ++I) { cin >> fv[I]; }
```

- Linked lists

```
#include <list>
std::list<smart_int> sample();
sample.push_back(value);
typedef std::list<smart_int>::iterator ilist;
for(ilist I=sample.begin();I!=sample.end();++I) {
  I->randomize();
}
sample.sort();
```

STL Containers continued

- Maps - associative container, sparse

```
#include <map>
std::map<packet, int> pstat;
… pstat[pkt]++; …
typedef map<packet,int>::iterator imap;
for(imap i=pstat.begin(); i!=pstate.end(); ++i) {
  cout << i->first.type
       << " occurred " << pstat->second << endl;
}//endfor
```

- Sets

```
#include <set>
enum BusState {Idle,Rst,SRd,SWr,MRd,MWr};
std::set<BusState> bs; bs.clear();
bs.insert(Idle);
if (bs.count(MWr) == 1) bs.erase(Idle);
```


boost::array Intro

- An ordinary array with STL extensions like vector
  - Doesn't carry the overhead of resizing that vector does
  - Complete array assignment
  - Range checks optional
- USAGE:

```
#include "boost/array.hpp"
boost::array<T,SIZE> VAR;
```

- EXAMPLE

```
using boost::array;
array<int,4> a = { (1,2,3,4) };
typedef array<int,4>::iterator iterator_t;
for(iterator_t i=a.begin();i!=a.end();++i){
  *i=f(*i) + a[2]; // silly equation using *i
}
```

Range checked Array

- **vector**<T> and **array**<T,N> classes both have range checking in the form of **.at()** method
  - Not quite as natural as using operator[]
- Easy to remedy with a derived class

```
template<typename T, int N>
class Array : public boost::array<T,N> {
public:
  Array(): array<T,N>() {}
  T& operator[](int i) { return at(i); }
  const T& operator[](int i) const {
    return at(i);
  }
};
```


boost::format Intro

- **printf** with argument checks & more…
- EXAMPLE

```
#include "boost/format.hpp"
cout << boost::format(

   "Hi %s! x=%4.1f :%d-th step\n"

 ) % "Toto" % 20.19 % 50 ;
```

```
Hi Toto! x=20.2 :50-th step
```

boost::format continued

- ```
  cout << boost::format(
  "%1% %3% %2% %1%\n") % "aa" % 'b' % 'c' ;
   //OUTPUT: "aa c b aa"
  ```
- ```
  boost::format fmt(
     "%|2$3x|:>%|1$=20|<%|30Tx|");
   string s = str(fmt % "The title" % 17);
   cout<<fmt.size()<<endl<<fmt.str()<< endl;
  ```

```
30
11:>        My title        <xxxxx
123456789^123456789^123456789^
```


XtremeEDA

boost::regex Intro

- Regular expressions for C++
  - grep, sed, perl, vim, emacs searching
  - Several varieties of expressions including perl
  - Allows for both search and replace
- More general than just character strings
  - Can search arrays of data for data patterns
- Lots of methods/syntax
  - We'll limit ourselves to simple string example
- `#include "boost/regex.hpp"`
- Link with `-lboost_regex`

XtremeEDA

boost::regex methods

- `boost::regex_match` determines if an expression matches an entire text
- `boost::regex_search` finds expression within a text
  - Most likely what you want to use
  - Allows identifying sub-matches
- `boost::regex_replace` makes replacements
  - Allows for sub-matches in replacement

boost::regex Example

```
#include "boost/regex.hpp"
string text("This is some text to search")
string::const_iterator text_beg = text.begin();
string::const_iterator text_end = text.end();
boost::regex expr("some text");
boost::match_results<string::const_iterator> rslt;
bool found = boost::regex_search
                (text_beg, text_end, rslt, expr)
if (found) cout << "Matched "
   << string(rslt[0].first,rslt[0].second)
   << " @ posn " << (rslt[0].first - text_beg)
   << " length " << (rslt[0].second - rslt[0].
   << endl;
```

What to search

Regular expression

The search

Where found

Boost Shared Pointers Intro

- Pointers are dangerous because it is easy to lose track of and create memory leaks
- Smart pointers solve this by providing garbage collection
- Six types

scoped_ptr	Simple sole ownership of single objects. Noncopyable.
scoped_array	Simple sole ownership of arrays. Noncopyable.
shared_ptr	Object ownership shared among multiple pointers
shared_array	Array ownership shared among multiple pointers
weak_ptr	Non-owning observers of an object owned by shared_ptr.
intrusive_ptr	Shared ownership of objects with an embedded reference count.

boost::shared Intro

- Shared pointers allow copying without worrying about dangling pointers. When reference count drops to zero, the object is destoyed.
  - Caveat: Dangerous if circularly linked (RARE)
- USAGE:
  - #**include** "**boost/shared_ptr.hpp**"
  - **boost::shared_ptr**<T> v1_ptr(**new** T);
  - **boost::shared_ptr**<T> v2_ptr;
  - v2_ptr.**reset**(**new** T);
  - v1_ptr = v2_ptr;
  - *v2_ptr = value;
  - **std::cout** << *v1_ptr << **std::endl**;
  - {**boost::shared_ptr**<TYPE> v3_ptr(**new** T);}

Normal pointers would create memory leaks

Questions

XtremeEDA