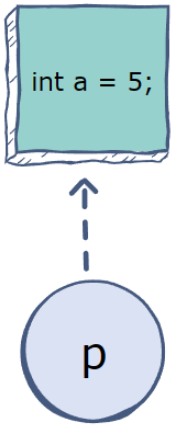
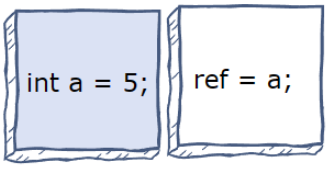
DCS Lab02

Basic of C++

- pointers and references
  - A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced with the * operator to access the memory location it points to.
  - A reference variable is an alias, that is, another name for an already existing variable.



The pointer variable **p** stores the address of the variable **a**; it "points" to the memory location of **a**.



Creating a reference to **a** just makes an alias for it; it does not "point" to **a** by storing its address in a separate memory location

- Template : Pass the data type as a parameter so that we don't need to write the same code for different data types.

```
template<typename T>
class TheClassName {
    // T can be treated as a type inside the class definition block
}

int main() {
    TheClassName<int> a;
    TheClassName<double> b;
    TheClassName<AnotherClassName> c;
}
```

- Operator Overloading : giving special meaning to an existing operator in C++ without changing its original meaning

```
struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display();
}
```

Reading and Writing Data with `sc_signal<T>`

There are two ways to read data from a `sc_signal`: you can call the member function `T& read()` or use the overloaded operator `()`.

```
sc_signal<int> s;
// method 1 - member function
std::cout << s.read();
// method 2 - operator overloading
std::cout << (s);
```

Similarly, there are two ways to write data to a `sc_signal`: you can call the member function `void write(const T&)` or use the overloaded operator `=`.

```
sc_signal<int> s;
// method 1 - member function
s.write(5);
// method 2 - operator overloading
s = 5;
```

Selecting Bits in `sc_signal` (Used in HW01 ALU)

Given an instruction:

```
sc_signal<sc_uint> instruction;
```

Idea: Since `sc_uint` provides methods like `range(int high, int low)` and overloaded operators, we can...

```
instruction.range(4, 0); // compile error
```

Reason for error: While `sc_uint` provides the ability to select bits, `sc_signal` does not. So, you need to first read the data from `sc_signal`, then use the methods or overloaded operators provided by `sc_uint`.

```
sc_signal<sc_uint> instruction;
// method 1 - member function / member function
instruction.read().range(4, 0);
// method 2 - member function / operator overloading
(instruction.read())(4,0)
// method 3 - operator overloading / member function
(instruction).range(4, 0)
// method 4 - operator overloading / operator overloading
(instruction)(4, 0)
```

Difference between `SC_CTOR(module)` and `SC_HAS_PROCESS(module)`

Both `SC_CTOR(module)` and `SC_HAS_PROCESS(module)` initialize the `SC_MODULE` by registering member functions of the class to the SystemC simulation kernel, enabling simulation processes to run.

- `SC_CTOR(module)` provides a default constructor, which only takes one argument - the module name.
- `SC_HAS_PROCESS(module)` does not provide a default constructor; the user needs to define the constructor.

So, when a module does not need parameters other than the module name for initialization, use `SC_CTOR(module)` to invoke the default constructor. When a module needs parameters other than the module name for initialization, use `SC_HAS_PROCESS(module)` and define the constructor with the required arguments.

```
#include <systemc.h>
```

```
SC_MODULE(MODULE_A) { // module without simulation processes doesn't need SC_CTOR
or SC_HAS_PROCESS
  MODULE_A(sc_module_name name) { // c++ style constructor, the base class is
implicitly instantiated with module name.
    std::cout << this->name() << ", no SC_CTOR or SC_HAS_PROCESS" << std::endl;
  }
};

SC_MODULE(MODULE_B) { // constructor with module name as the only input argument
  SC_CTOR(MODULE_B) { // implicitly declares a constructor of
MODULE_A(sc_module_name)
    SC_METHOD(func_b); // register member function to simulation kernel
  }
  void func_b() { // define function
    std::cout << name() << ", SC_CTOR" << std::endl;
  }
};

SC_MODULE(MODULE_C) { // pass additional input argument(s)
  const int i;
  SC_HAS_PROCESS(MODULE_C); // OK to use SC_CTOR, which will also define an un-
used constructor: MODULE_A(sc_module_name);
  MODULE_B(sc_module_name name, int i) : i(i) { // define the constructor
function
    SC_METHOD(func_c); // register member function
  }
  void func_c() { // define function
    std::cout << name() << ", additional input argument" << std::endl;
  }
};

int sc_main(int, char*[]) {
  MODULE_A module_a("module_a");
  MODULE_B module_b("module_b");
  MODULE_C module_c("module_c", 1);
  sc_start();
  return 0;
}
```

Difference between `SC_METHOD(func)` and `SC_THREAD(func)`

- `SC_METHOD(func)` does not have its own thread, does not consume simulation time, cannot call `wait()`, and establishes dynamic sensitivity using `next_trigger()`, executing when triggered. Thus, it can repeatedly execute without using an infinite loop.
- `SC_THREAD(func)` has its own thread, may consume simulation time, establishes dynamic sensitivity using `wait()`, and needs an infinite loop.
- Another type `SC_CTHREAD(func, event)` is a special form of `SC_THREAD(func)` with only static sensitivity, typically using a clock event as the sensitivity event.

Simulation Process	SC_METHOD	SC_THREAD	SC_CTHREAD
Execution	When triggered	Always execute	Always execute
Suspend time	No	Yes	Yes
Static sensitivity	By sensitive list	By sensitive list	By clock event
Dynamic sensitivity	`next_trigger()`	`wait()`	X
Infinite Loop	No	Yes	Yes
Applied model	RTL, synchronize	Behavioral	Clocked behavior

```
#include <systemc.h>
SC_MODULE(PROCESS) {
  sc_clock clk; // declares a clock
  SC_CTOR(PROCESS) : clk("clk", 1, SC_SEC) { // instantiate

 a clock with 1sec periodicity
    SC_METHOD(method); // register a method
    SC_THREAD(thread); // register a thread
    SC_CTHREAD(cthread, clk); // register a clocked thread
  }
  void method(void) { // define the method member function
    // no while loop here
    std::cout << "method triggered @ " << sc_time_stamp() << std::endl;
    next_trigger(sc_time(1, SC_SEC)); // trigger after 1 sec
  }
  void thread() { // define the thread member function
    while (true) { // infinite loop make sure it never exits
      std::cout << "thread triggered @ " << sc_time_stamp() << std::endl;
      wait(1, SC_SEC); // wait 1 sec before execute again
    }
  }
  void cthread() { // define the cthread member function
    while (true) { // infinite loop
      std::cout << "cthread triggered @ " << sc_time_stamp() << std::endl;
      wait(); // wait for next clk event, which comes after 1 sec
    }
  }
};

int sc_main(int, char*[]) {
  PROCESS process("process"); // init module
  std::cout << "execution phase begins @ " << sc_time_stamp() << std::endl;
  sc_start(2, SC_SEC); // run simulation for 2 second
  std::cout << "execution phase ends @ " << sc_time_stamp() << std::endl;
  return 0;
}
```

Result

```
// module instantiation finished
execution phase begins @ 0 s
// all processes are triggered once
method triggered @ 0 s
thread triggered @ 0 s
cthread triggered @ 0 s
// all processes are triggered again after 1 sec, using different methods
method triggered @ 1 s
thread triggered @ 1 s
cthread triggered @ 1 s
// simulation ends after 2 seconds
execution phase ends @ 2 s
```

Usage of `sc_port`

A port is a pointer to a channel, which needs to be connected (bound) to a channel before it can read or write data. When reading data, you need to dereference the port to access the channel it points to, and then use the channel's member functions `write()` and `read()` to access the data.

```
#include <systemc.h>
#include <string>

SC_MODULE(MODULE1) {                    // top-level module
    sc_port<sc_signal_out_if<std::string>> p; // port
    SC_CTOR(MODULE1) {
        SC_THREAD(writer);
    }
    void writer() {
        std::string text = "DCS_Lab02_"; // init value
        while (true) {
            std::string write_data = text +
std::to_string((int)sc_time_stamp().to_seconds());
            p->write(write_data); // writes to channel through port
            std::cout << sc_time_stamp() << ":" << name() << " writes to channel,
string=" << write_data << std::endl;
            wait(1, SC_SEC);
        }
    }
};

SC_MODULE(MODULE2) {
    sc_port<sc_signal_in_if<std::string>> p;
    SC_CTOR(MODULE2) {
        SC_THREAD(reader);
        sensitive << p; // triggered by value change on the channel
        dont_initialize();
    }
    void reader() {
        while (true) {
            std::cout << sc_time_stamp() << ":" << name() << " reads from
channel, string=" << p->read() << std::endl;
            wait(); // receives from channel through port
        }
    }
```

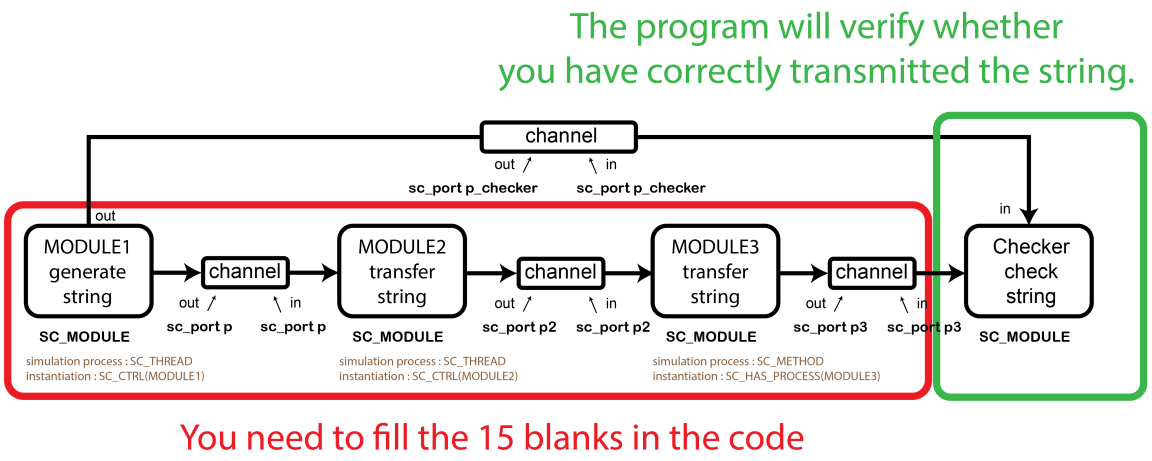
```
};

int sc_main(int, char *[]) {
    MODULE1 module1("module1"); // instantiate module1
    MODULE2 module2("module2"); // instantiate module2
    sc_signal<std::string> s;   // define channel outside module1 and module2
    module1.p(s);               // bind module1's port to channel, for writing
purpose
    module2.p(s);               // bind module2's port to channel, for reading
purpose
    sc_start(2, SC_SEC);
    return 0;
}
```

Result

```
0 s:module1 writes to channel, string=DCS_Lab02_0
0 s:module2 reads from channel, string=DCS_Lab02_0
1 s:module1 writes to channel, string=DCS_Lab02_1
1 s:module2 reads from channel, string=DCS_Lab02_1
```

Lab02 Specifications



Students need to complete 15 code blanks. Module1 generates a string every 5 seconds and passes it to Module2, then Module3, and finally to the checker. The transmission is limited to using `sc_port` pointers to channels. Module3 is required to use

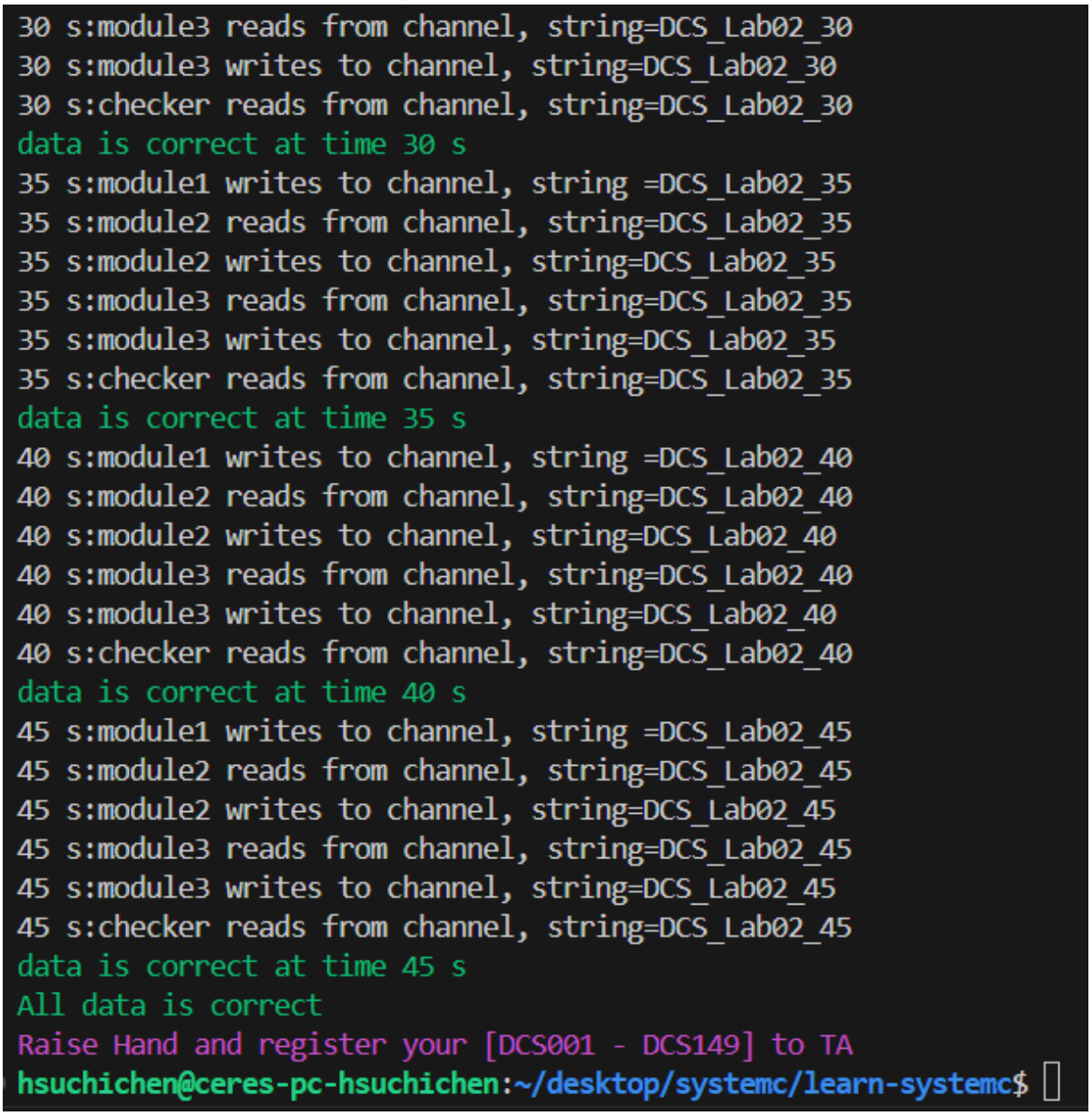
1. simulation process: `SC_METHOD`
2. instantiation: `SC_HAS_PROCESS`

This is to understand the difference between `SC_METHOD` and `SC_THREAD`, and between `SC_HAS_PROCESS` and `SC_CTRL`.

The program will pass the strings generated by Module1 directly to the checker, and compare them with the strings generated by the student's code to check if the transmission is correct.

Expected Result

Each channel read and write will output data.



Lab02 Tips

- (1, 2, 10, 11) How to write to / read from a channel? Dereference the port and call member functions `write()` and `read()` to write the generated string to the channel. See Usage of `sc_port` for syntax.
- (3) How to wait on events in sensitivity list? See Combined Events for syntax.
- (4) How to define a port? See Usage of `sc_port` for syntax.
- (6) How to declare `SC_HAS_PROCESS`? See Difference between `SC_CTOR(module)` and `SC_HAS_PROCESS(module)` for syntax.
- (7, 12) How is `SC_METHOD` triggered? See Difference between `SC_METHOD(func)` and `SC_THREAD(func)` for syntax.
- (8) When does `p2`'s value change trigger `SC_METHOD(transfer)`? How to avoid it being called? See lecture notes, slide 2, page 47.
- (9) Is `SC_METHOD(transfer)` called during initialization? How to prevent it from being called? See lecture notes, slide 2, page 47.

- (13, 14, 15) How to bind ports to channels? See Usage of `sc_port` for syntax.

Command List

1. Extract files from TA's directory

```
tar -xvf ~DCSTA01/Lab02.tar
```

2. Compile & Run SystemC source code

```
./01_systemc
```

3. Clean SystemC executable file

```
./09_clean
```

4. Hand in `port2port.cpp` to E3 platform (If demoed by TA during class and already registered, no need to submit to the E3 platform)

Grading Criteria

- Completed before 6 PM on the same day and raised your hand to inform the TA of your server number `[DCS001 - DCS155]`: 100 points. After registering, you can check the Lab02 Registration Form Link to confirm if the TA has registered you. **If demoed by TA during class and already registered, no need to submit to the E3 platform**.
- Submitted to E3 platform before 11:59 PM on the same day: 80 points.
- Submitted the next day: 60 points, and so on.