CS-472: Team of Teaching Assistants

Andrea Costamagna
(`andrea.costamagna@epfl.ch`)

- PhD candidate at Integrated Systems Laboratory (LSI), EPFL
- Research interests:
  - Logic optimization for high-performance digital circuits.

Mingfei Yu (`mingfei.yu@epfl.ch`)

- PhD candidate at LSI, EPFL
- Research interests:
  - EDA for faster secure computation;
  - Quantum compilation.

CS-472: Exercise Sessions, Homework and Project

- Exercise Problem Sets
  - Pen-and-paper exercises
  - To prepare you for the exams
  - Not graded, no need to hand in
  - Reference solution will be released at the beginning of the following week
- Homework and Project
  - Programming assignments (mostly in C++)
  - Graded; one or two weeks time
- Exercise Sessions
  - When? Thursdays 11:15 - 13:00
  - What?
    1. Discussion of the solutions of the previous week's exercise problems
    2. Release of new exercise problem set
    3. Free time to do homework/exercises and ask individual questions
    4. Quick practical tutorials (when appropriate, e.g., today)

Skills and background knowledge you will need (Moodle post)

- C++
  - Basic to intermediate programming ability is assumed.
  - A quick tutorial is given today + HW0 for warming-up practice.
  - Learn from given example code & online resources.
  - Efficiency & quality of code are not emphasized, but must be functionally correct.
- VHDL
  - You will need to write/modify a bit of VHDL code in HW2.
  - A quick tutorial will be given.
- Graph theory
  - Chapter 2 ("Background", thus will not be taught in lectures) in the textbook.
  - A brief review is given today + exercise problems.
- Command line and git
  - Connecting to servers, basic linux commands, editing text files with command line (vim, emacs etc.), compiling with make, etc.
  - Project is hosted on GitHub and submission is via pull requests.
  - Simple instructions will be given. Try to search and learn on your own.

C++ Tutorial for CS-472

What you need to know & self-learning tips

Mingfei Yu

Original slides by Siang-Yun (Sonia) Lee

LSI, EPFL

September 19, 2024

Basic Concepts

The programming language `C++`...

- ...is an **object-oriented** language. `class Dog{...}; Dog charlie; charlie.bark();`
- ...is a **compiled** language.
- ...is **strongly typed**. `int x = 10; float y = 3.14;`
- ...provides possibility of **low-level memory manipulation**.
- ...comes with many well-implemented **standard libraries**. `std::vector, std::sort`

Why `C++`?

- Gain more control in *what exactly is executed in the CPU*.
- Customization (and hacks) to increase **efficiency** and to reduce **memory usage**.
- Many EDA tools are written in `C/C++`.

C++ Basics (1)

Comments and basic syntax.

```cpp
int main()
{
  // single-line comment
  /* multiple-line
  comment */
  int x = 10; // Remember the semicolon!
  for ( int i = 0; i < 100; i++ )
  {
    if ( x > 0 )
    { // Brackets are not mandatory for single-line body,
      x--;
    } // but it's a good habit!
  }
}
```

C++ Basics (2)

Command-line arguments and standard libraries.

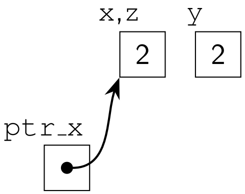
```
int main( int argc, char* argv[] )
{
  // argc = number of command-line arguments
  // argv = array of command-line arguments
  // argv[0] is always the executable name

  std::vector<int> v;
  // vector : size-dynamic array
  v.push_back( std::stoi( argv[1] ) );
  // push_back : add an element at the end
  // stoi : convert from string to int
  std::cout << "first element is " << v.at(0) << std::endl;
  // cout : print
  // endl : end line
}
```

In a command-line interface: `g++ main.cpp -o example; ./example 3`

C++ Basics (3-1)

Address, pointers and references.

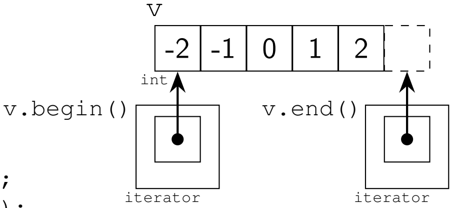



```cpp
int main()
{
  int x = 2;
  int* ptr_x = &x;
  // int* : a pointer to an int
  // &x : take the address of x
  int y = *ptr_x;
  // *ptr_x : "dereference" the pointer (get back x from its address)
  int& z = *ptr_x;
  // int& : a reference (alias) of an int
  // i.e., z is the same thing as *ptr_x, which is the same thing as x
  *ptr_x += 1;
  std::cout << "x = " << x << ", y = " << y << ", z = " << z << std::endl;
  // x = 3, y = 2, z = 3
}
```

C++ Basics (3-2)

The C++ way: Iterators

- Roughly the same as pointers conceptually.
- Compiler recognizes it as an **object**, but not an **address** (numeric value) → type safety.
- Iterates through **a range**.




```cpp
int main()
{
  std::vector<int> v = { -2, -1, 0, 1, 2 };
  std::vector<int>::iterator it = v.begin();
  while ( it != v.end() )
  {
    *it = -(*it); // use * to get or operate on the real content
    ++it; // move forward
  }
}
```

C++ Basics (4-1)

Classes and objects.

```
class Student {
public: // public member functions are available from outside
  Student( std::string name, int grade ) // constructor
    : _name( name ), _grade( grade ) // initialize data members
  { std::cout << "A student is created!\n"; }

  ~Student() { std::cout << "A student is deleted!\n"; } // destructor

  std::string get_name() const
  { return _name; }

  void set_grade( int new_grade )
  { _grade = new_grade; }

private: // data members are usually private
  std::string _name;
  int _grade;
}; // Remember the semicolon!
```

C++ Basics (4-2)

Classes and objects.

```cpp
int main()
{
  Student s1( "John", 90 ); // instantiate an object
  s1.set_grade( 95 );
  // call a member function with an object using a dot

  Student* ptr_s2 = new Student( "Mary", 80 ); // another way of instantiation
  std::cout << "student's name is " << ptr_s2->get_name() << std::endl;
  // when calling with a pointer, use ->

  delete ptr_s2;
  // if you instantiate with "new", then you need to explicitly "delete" it
}
```

C++ Basics (5-1)

Scope of variables.

```cpp
void func1( int x ) // x is pass by copy
{
  x += 1; // does not affect the x in main
  int z; // func1-wide variable
}

int main()
{
  int x = 1; // main()-wide variable
  { // every pair of {} creates a local scope
    // (looping, conditioning, function, or simply brackets like this)
    int y; // {}-wide variable
  } // y is deleted at this point (call destructor)

  func1( x );
}
```

C++ Basics (5-2)

Scope of variables.

```
void func2( int& x ) // x is pass by reference
{
  x += 1; // affects the x in main
  // x += y; // this will not work
}

int main()
{
  int x = 1;
  {
    int y = 2;
    func2( x );
    func2( y ); // but this works
  }
  std::cout << "x in main is " << x << std::endl;
  // x in main is 2
}
```

C++ Basics (5-3)

Scope of variables.

- If you write a stand-alone function, you need to pass all needed variables as its arguments.
- If you write a **member function** of a class, all data members are accessible – useful when you have lots of "global variables".
- These are all local variables. Real global variables (defined outside of any function or class) are usually not recommended.

More Advanced C++ Features

- `template`: Provides flexibility in function and class implementations.

```
template<class T>
T add_three( T a, T b, T c )
{ return a + b + c; }

int main()
{
  add_three<int>( 1, 2, 3 );
  // substitute "T" with "int" in function definition
}
```

- `namespace`: To avoid collision in function or class names.

```
namespace dtis { template<class T> class vector{ /* ... */ }; }
int main()
{
  using namespace std; // std will be the default namespace
  vector<int> v1; // std::vector (standard library)
  dtis::vector<int> v2; // our own implementation
}
```

Understanding a Piece of C++ Code

Basic steps

1. Understand the **data structure**.
2. Locate the entry point of the program (`int main()`) and of the main algorithm (typically a function call).
3. Focus on the big picture before diving into details and corner cases/terminating conditions.

Pro-tips

- Print out stuff to quickly know which parts of the code are executed and in which order.
- Try to "execute by hand" the core part line by line with a toy example. If you don't know what an expression evaluates to, print it out!
- If you see...
  - ...an object of a custom class: Find the class definition. Pay attention to what **data members** it has.
  - ...a **standard library** object: Search for online documentation (`cppreference.com`).

Writing C++ Code

- Learn from some example code.
- Leverage the standard libraries (e.g., `std::vector`, `std::set`, `std::map`, `std::pair`, `std::sort` ...).
- Plan your data structure.
- Break down the task into simple steps. Write one step, **compile it**, and do simple tests if possible.
- Write assertions (e.g. `assert(x > 0);`) whenever there are explicit or implicit assumptions, or when you expect the computation result to have some properties.
- The basic stuff: Use a good editor. Properly indent the code (though not mandatory). Give variables, classes and functions reasonable names. Write comments for yourself.

Debugging C++ Code

- Compilation errors
  - Ask `Google/ChatGPT` about the error message.
  - Be aware of the `const` qualifiers.
- Segmentation faults
  - Means: illegal memory access.
  - Usually: double free, index out of bounds.
  - Add print-outs to locate the point where it segfaults (remember to add `cout.flush();`).
  - Use a debugger (`gdb`, `lldb`, etc.).
- Assertion fails and incorrect results
  - Print variable values (using a debugger may be convenient).
  - Execute the example by hand and compare with the actual computation results.
  - Add more assertions (maybe it has been wrong way earlier).
  - Explain your code to a rubber duck ;-)

Writing **Good** `C++` Code

(Not a requirement for the course)

- Avoid unnecessary copying. (pass by copy vs. pass by reference)
- Understand the underlying data structure and the complexities of its operations. (e.g. `std::vector` vs. `std::map`)
- Distinguish what are decided (evaluated/instantiated/substituted) at **compile-time** and what are known only at run-time.
- Distinguish public and private data members and member functions. (Does it need to be public?)
- Add the `const` qualifier whenever appropriate.
- Avoid duplicated code whenever possible.