EL2310 – Scientific Programming

Lecture 13: Introduction to C++

Ramviyas Parasuraman (ramviyas@kth.se)
Credits: Andrej Pronobis (U.Washington), Florian Pokorny (UC Berkeley)

KTH - Royal Institute of Technology

Overview

Lecture 13: Introduction to C++

- Differences between C and C++
- Printing and User Input
- Namespaces
- References and Pointers
- Allocating Memory Dynamically

Introduction to Object Oriented Paradigm

- More on Object Oriented Programming
- Classes

Course so far

- MATLAB: Using program to perform some tasks
- C: Learning how to program

Rest of the course

- C++
  - Writing extendable (modular) programs in C++
  - Object Oriented Programming
  - Using other people's code
  - Modifying/Extending other people's code
  - Writing re-usable code

Today

- Basics in C++
- Introduction to OOP

What is C++?

- Developed by Bjarne Stroustrup starting from 1979 at Bell Labs
- Adds object oriented features (e.g. classes) to C
- Initially named: C with Classes; then renamed to ”C++” (guess why?)
- Influenced many other languages: C#, Java
- The C++ standard library incorporates:
  - The C standard library with small modifications
  - STL (Standard Template Library)
- Constantly developed: C++11 (2011), C++14 (2014)
- P.S: Objective-C uses another approach to adding classes to C

C++ subsumes C

- You can use all you learned in C in C++ as well
- Some constructs/syntax have a C++ version

File naming conventions

- We named files in C as .c (source) and .h (header)
- In C++, the ending is typically .cc or .cpp for source files and .h, .hh or .hpp for header files
- In this course we will use .cpp and .h

C++ Compiler

- `g++` and `gcc`: `g++` is specific to C++ by (auto) linking to std C++ libraries whereas `gcc` decides based on the file extension (.c/.cpp)
- Usage and command line options for `g++` are the same as for `gcc`
- Make sure you know how to use `make` for this part of the course!

Comments in C++

- Multi-line comments as in C, i.e. `/* ...*/`
- Single-line comments using `//`
- : Example:

```
int main() {
  // This is a single line comment
/* This comment extends to
multiple lines */   ...
}
```

Basic data types

- All data types from C can be used. Plus some more, e.g.
- `bool`: boolean value `true/false`
- `string`: "real" string (use `#include <string>`)

Declaration of variables

- You no longer need to declare the variable at the beginning of the function (scope), as was the case for pre C99
- Rule of thumb: declare variables close to where they're used.
- For instance:

```
for(int i=0;i<N;i++){...}
```

  `i` only defined within loop
- Use specific names for counters, e.g. `i,j,k,...`

Printing to Screen

- In C++ we use *streams* for input and output using the `<iostreams>` library
- Output is handled with the stream `cout` and `cerr`, using the `<<` operator
  Ex: `cout << "Hello world";`
- To add a line feed use the "\n" as in C or the special `endl`

```
cout << "Hello world\n" ;
cout << "Hello world" << endl;
```

- All basic data types have the ability to add themselves to a stream for printing

Printing to screen cont'd

- You can mix data types easily
- In C: `printf("The value is %d\n", value);`
- In C++: `cout << "The value is " << value << endl;`
- The stream `cerr` is the error stream
- Compare `stdout` and `stderr` in C

Formatting output

- Just like in C you can format the output in a stream
- You can use

  `width` number of characters for output to fill
  `precision` number of digits
  `fill` pad with a certain character

- Syntax:

```
cout.precision(4);
cout.width(10);
cout.fill('0');
cout << 12.3456789 << endl;
```

- Will output 0000012.35
- Default precision=6, fill=' ' (space)

Getting input from the user

- streams is also used to get input from console
- Use the `cin` stream
- Ex:

```
int value;
cin >> value;
```

- Using `cin` will flush the `cout` stream

Reading strings

- When reading with `cin`, the inputs are separated by spaces
- Ex: `cin >> a >> b >> c;`
- If you want to read an entire line, use `getline`
- Ex:

```
string line;
getline(cin, line);
cout << "The input was " << line << endl;
```

Hello KTH in C++

```
#include <iostream>
int main ()
{
  std::cout << "Hello KTH!";
  return 0;
}
```

Hello KTH in C++...

- `<iostream>` replaced `<stdio.h>`
- Standard C++ header files are included without the suffix (no .h at the end)
- Here the `std` namespace is used, where `cout` is found.
  Let's know more about namespaces now!

Task 1

- Write a program that reads the name and age of a person
- It should then print this info on the screen

Namespaces

- In C all function share a common `namespace`
  - This means that there can only be one function for each function name
- In C++ functions can be placed in specific namespaces
- Syntax:

```
namespace NamespaceName {
  void fcn(); ...
}
```

Accessing functions in a namespace

- To access a function `fcn` in namespace `A`,
  `A::fcn`
- This way you can have more than one function with the same name but in different namespaces

Using namespace

- Specifying the namespace all the time == lose time in typing
  ```
  std::cout << "Who likes typing?" << std::endl;
  ```
- Solution: extending a specific namespace in a program,
- E.g.
  ```
  using namespace std
  cout << "OK" << endl;
  cout << "Now it feels much better!" << endl;
  ```
- But avoid using this in header files

Task 2

- Write a program to test the idea of namespaces
- Define two functions `fcn();` inside namespaces `A` and `B`

References

- “Constrained” pointers and “(a bit) safer” references
- Compare

```
int a;                      int a;
int *pa = &a;               int &ra = a;
int *pa = NULL;             -
*pa = 10;                   ra = 10;     => a==10
int b;                      int b;
pa = &b;                    -
int *pc;                    -
pc = pa;                    -
```

Pointers vs References

- References need to be assigned when constructed
- Ex: This is not allowed

```
int &x;
int y;
x = y;  (assigned too late)
```

- Try to use references within functions
- Pointers can be re-assigned anytime. Use pointers in your algorithms and computations

Passing Arguments by Reference

- Standard function calls are by value
- Value of the variable is copied into the function
- Pointers offered a way in C to do call by reference
- Call by reference avoids the need to copy all the data
- Ex: Not so good to copy an entire 10Mpixel image into a function, better to give a reference to it (i.e. tell where it is)
- In C++ we can use references

Passing Arguments by Reference, Cont'd

- Declaration: `void fcn(int &x);`
- Any changed to `x` inside `fcn` will affect the parameter used in the function call
- Ex:

```
void fcn(int &x)
{
  x = 42;
}

int main()
{
  int x = 1;
  fcn(x);
  cout << "x=" << x << endl;
}
```

- Will change value of `x` in the scope of `main` to 42

Dynamic Memory Allocation in C++

- In C we used `malloc` and `free`
- In C++ the `new` and `delete` operators are used
- Ex:

```
int *p = new int;
*p = 42;
...
delete p;
```

DMA for Arrays

- If you allocate an array with `new` you need to delete with `delete []`
- Ex:

```
int *p = new int[10];
p[0] = 42;
delete [] p;
```

- A typical mistake: forgotten []

The Object-Oriented Paradigm

Motivation:

- We are trying to solve complex problems
  - Complex code with many functions and names
  - Difficult to keep track of all details
- How can we reduce the complexity?
  - Grouping related things
  - Abstracting things away
  - Creating hierarchies of things
- Advantages:
  - Re-usable and reliable code
  - Ease of debugging

Key Concepts of OOP

- Classes (types)
- Instances (objects)
- Methods (actions)
- Interfaces
- Encapsulation
- Polymorphism
- Inheritance
- Access protection - information hiding

Car example

Object Oriented Programming (OOP)

- Encapsulation
  - Bundle data and the code to process it
  - Can create a “black-box” with well defined interface
  - Hiding the inside means you can not change the inside
  - this bundle or box is the *object*
- Polymorphism
  - “one interface, multiple methods”
  - Can have the same interface for many classes that do the same thing

Object Oriented Programming (OOP)

- Encapsulation
  - Bundle data and the code to process it
  - Can create a “black-box” with well defined interface
  - Hiding the inside means you can not change the inside
  - this bundle or box is the *object*
- Polymorphism
  - “one interface, multiple methods”
  - Can have the same interface for many classes that do the same thing

Object Oriented Programming (OOP)

- Inheritance
  - Support for hierarchies (most knowledge can be structured by hierarchical classifications)
  - Ex: A car is a motor vehicle which is a vehicle which is a transportation system which is a ...
  - Subclass to inherit the properties of the base class

Inheritance

- Inheritance is a way to show a relation like “is a”
- Ex: A car is a vehicle
- A car inherits many of its properties from being a vehicle
- These same properties could also be inherited by a truck or a bus
- Syntax: `class Car : public Vehicle` to tell that Car inherits from Vehicle

Inheritance vs Aggregation

- Inheritance correspond to “is a” relations
- Ex:

```
class Car :  public Vehicle ...
```

- Aggregation to “has a”
- Ex:

```
class Car {
...
Person m_Owner;
```

Constructors

- Constructor is the first thing to run while instantiating a class
- If you do not specify a constructor in the initialization list the default constructor will be called

Inheritance and Constructors

- If you have three classes A, B and C, where B inherits from A and C inherits from B
- When you create C the constructor from the base classes (B and A) will be run first
- Execution order
  1. Initialization list for A runs. Body of A constructor runs
  2. Initialization list for B runs. Body of B constructor runs
  3. Initialization list for C runs. Body of C constructor runs

Function overloading

- You can have many functions with the same name but with different parameter declarations
- In C we have `int abs(int)` and `double fabs(double)`
- Need to have different names in C
- In C++ you can have the same name!
- You heard something like this before, right?

Classes

- A `class` is an “extension” of a `struct`
- A class can have both data member and function members (methods)
- Classes bring together data and operations related to that data
- Like C structs, classes define new data types
- Unlike structs, they also define how operators work on the new types

Class definition

- Syntax:

```
class ClassName {
public:
  void fcn();
private:
  int m_X;
}; // Do not forget the semicolon!!!
```

- `m_X` is a member data
- `void fcn()` is a member function
- `public` is an access specifier specifying that everything below can be access from outside the class
- `private` is an access specifier specifying that everything below is hidden from outside of the class

Access specifiers

- There are three access specifiers:
  - `public`
  - `private`
  - `protected`
- No access specifier specified $\Rightarrow$ assumes it is `private`
- Data and function members that are `private` cannot be accessed from outside the class
- Ex: `m_X` above cannot be accessed from outside
- `protected` will be discussed later

C++ Structs

- C++ also uses `struct`
- In C++ `struct` is just like a class (much more than the C `struct`!)
- The only difference is the default access protection:

```
class Name {
  int m_X; // Private
};
struct Name {
  int m_X; // Public
};
```

Classes and Objects

- Classes define data types
- Objects are instances of classes
- Objects correspond to variables
- Instantiating a class (Declaring an object):
  `ClassName variableName;`

Classes and Namespace

- The class defines a namespace
- Hence function names inside a class do not name-clash with other functions
- Example: the member variable `m_X` above is fully specified as `ClassName::m_X`

Constructor

- Constructor is a special kind of method.
- The constructor tells how to “setup” the objects
- The constructor that does not take any arguments is called the *default constructor*
- When an object of a certain class is created (instantiated), the so-called *constructor* is called first
- The constructor has the same name as the class and has no return type

```
class A {
public:
  A() {}
};
```

Constructor

- Some data types cannot be assigned, only initialized, e.g. references
- These data members should be initialized in the *initializer list* of the constructor
- Try to do as much of the initialization in the initialization in the list rather than using assignment in the body of the constructor
- Variables are initialized in the order they appear in the list

```
class A {
public:
  A():m_X(1) {}
private:
  int m_X;
};
```

Constructor Example

```
class A {
public:
  A():m_X(1) {}
  int getValue() { return m_X; }
private:
  int m_X;
};
A a;
std::cout << a.getValue() << std::endl;
```

Multiple Constructors

- You can define several different constructors
- ```
  class MyClass {
  public:
    MyClass():m_X(1) {}
    MyClass(int value):m_X(value) {}
    int getValue() { return m_X; }
  private:
    int m_X;
  };
  MyClass a; // Default constructor
  MyClass aa(42); // Constructor with argument
  std::cout << a.getValue() << std::endl;
  std::cout << aa.getValue() << std::endl;
  ```
  ```