

# Intro to C++

*R. A. Angryk*

## Courses Evaluation

- Course Evaluation Website is open!
- Please, provide your feedback.
- The survey will close on Friday!

<http://www.cs.montana.edu/survey/>

## Dr. Keller on Campus – this Friday!

- 11:00am - Computer Science Seminar (in 108 EPS, MSU-Bozeman):**  
**Soft Computing for Sensor and Algorithm Fusion**
- 3:00pm - COE seminar (in 101 RH, MSU-Bozeman):**  
**Spatial Reasoning: From Sketch-to-Text Towards Text-to-Sketch**

<http://cirl.missouri.edu/Keller/>

## Dr. Keller (1)

- James M. Keller received the Ph.D. in Mathematics in 1978. He holds the University of Missouri Curators' Professorship in the Electrical and Computer Engineering and Computer Science Departments on the Columbia campus. He is also the R. L. Tatum Professor in the College of Engineering.
- His industrial and government funding sources include the Electronics and Space Corporation, Union Electric, Geo-Centers, National Science Foundation, the Administration on Aging, The National Institutes of Health, NASA/JSC, the Air Force Office of Scientific Research, the Army Research Office, the Office of Naval Research, the National Geospatial Intelligence Agency, and the Army Night Vision and Electronic Sensors Directorate.

## Dr. Keller (2)

- Jim is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) for whom he has presented live and video tutorials on fuzzy logic in computer vision, is an International Fuzzy Systems Association (IFSA) Fellow, is a national lecturer for the Association for Computing Machinery (ACM), is an IEEE Computational Intelligence Society Distinguished Lecturer, and is a past President of the North American Fuzzy Information Processing Society (NAFIPS). He received the 2007 Fuzzy Systems Pioneer Award from the IEEE Computational Intelligence Society. He is currently the Vice President for Publications of the IEEE Computational Intelligence Society.

## Outline

- C vs. C++
- History lesson
- Data Abstraction
- Object-Oriented Programming
- Some useful C++ syntax features
- Structures vs. Classes

7

## Be aware...

- C and C++ are portable languages
- Portability
  - C and C++ programs can run on many different computers
- Compatibility – a different story...
  - Many features of current versions of C++ not compatible with older implementations

8

## History of C

- Evolved from two other programming languages
  - BCPL and B (“Typeless” languages)
- Dennis Ritchie (Bell Laboratories)
  - Added data typing, other features
- Development language of UNIX
- Hardware independent
  - Portable programs
- 1989: ANSI standard
- 1990: ANSI and ISO standard published
  - ANSI/ISO 9899: 1990

9

## History of C++ (1)

- Extension of C (C++ for “A better C”)
- Early 1980s: Bjarne Stroustrup (Bell Laboratories)
- Provides capabilities for object-oriented programming
  - Objects: reusable software components → Model items in real world
  - Object-oriented programs → Easy to understand, correct and modify
- Hybrid language
  - C-like style
  - & Object-oriented style

10

## History of C++ (2)

- Bjarne Stroustrup
  - 1980: “C with Classes”
    - Improve program structure (Simula67)
    - Maintain run-time efficiency
    - Support rather than *enforce* effective programming techniques
- 1985: C++ 1.0
- 1995: Draft standard

11

## History of C++ (3)

- ISO/IEC 14882**
- 1998: International standardization**
- New type *bool*
  - *static\_cast*, *dynamic\_cast* etc
  - “Standard Library” (STL)
    - Generic containers
    - Generic algorithms
  - Compatibility with ANSI C
  - Greater Portability

12

## Standard Template Library (STL) – (1)

- **C++ programs**
  - Built from pieces called classes and functions
- **C++ standard library**
  - Rich collections of existing classes and functions
- **“Building block approach” to creating programs**
  - “Software reuse”

13

## Standard Template Library (STL) – (2)

- **C++ library of container classes, algorithms, and iterators**
- **it provides many of the basic algorithms and data structures of computer science**
- **a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template.**
- **You should make sure that you understand how templates work in C++ before you use the STL.**

14

## What is C++ ?

- **General purpose, biased towards systems programming:**
  - A better C
  - *Supports* data abstraction
  - *Supports* object-oriented programming
  - *Supports* generic programming

15

## Data Abstraction (ENCAPSULATION)

- **Define program entity in terms of related data.**
- **Define operations on entity.**
- **Separate implementation of program from its data structures (data hiding via object interfaces)**
- **Program relies more on abstract properties of classes, than on properties of particular objects**

16

## Object-Oriented ... O-O Programming vs. O-O Language

- **Extends data abstraction**
- **Facilitates code reuse through *inheritance***
- **Runtime *dynamic binding***
- **C++ is not “truly object-oriented”**
  - hybrid approach
  - Not everything is a class

17

## A few major domains of C++ Applications

- **Operating system kernels and components**
- **Device drivers**
- **Real-time / deterministic systems**
- **Critical systems**
- **Parallel computing**
- **Numerical calculations**
- **3D Games and Graphics.**

18

## A better C ...

- Symbolic constants
- Inline code substitution
- Default function arguments
- Function and operator overloading
- References
- Namespace management
- Exception Handling
- Templates

19

## C++ Syntax Features (1)

*Something you don't have in C*

- A reference is an alias (an alternate name) for an object.

```
int a = 1; int c = 10;
int& b = a;
int& d; // error
int* dd; // ok
```

You can't separate the reference from the referent

- References are frequently used for pass-by-reference

```
void swap(int& i, int& j) {
    int tmp = i;
    i = j;
    j = tmp;
}
```

Use references when you can, and pointers when you have to

```
int main() {
    int x, y;
    // ...
    swap(x,y);
}
```

20

## C++ Syntax Features (2)

- Symbolic constants

```
const int arraySize = 10;
int a[arraySize];
```

- Inline functions

- A C++ compiler generates a copy of an inline function in place to avoid function call.
- Good for small functions.

```
inline float cubeFun (const float s) {
    return s*s*s;}

```

- Default function arguments

```
void foo (int a, int b = 0){..}
int a, b;
foo (a); /* called foo function
with arguments (a, 0) */
```

21

## C++ Syntax Features (3)

```
for( int i = 0; i < maxi; i++)
{
    // statement(s)    <--- FOR LOOP BODY
}
```

- Note type inside the loop control.
- **i** is undefined outside the loop.

C++ has many other Syntax Features...

22

## Basics of a Typical C++ Environment

- Input/output

- cin

- Standard input stream
- Normally keyboard

- cout

- Standard output stream
- Normally computer screen

- cerr

- Standard error stream
- Display error messages

23

## Simple C++ program (1)

```
1 // myFirst.cpp
2 #include <iostream>
3
4 // function main begins program execution
5 int main()
6 {
7     std::cout << "Welcome to C++!\n";
8     return 0;
9 } // end function main
```

Preprocessor directive to include input/output stream header file <iostream>.

Stream insertion operator.

Name cout belongs to namespace std.

Welcome to C++!

24

### A Simple C++ Program: Printing a Line of Text

- Standard output stream object
  - `std::cout`
  - Standard output stream (`cout`) "connected" to standard space (`std`)
  - Binary scope resolution operator (`::`)
  - `<<`
    - Stream insertion operator
    - Value to right (right operand) inserted into output stream
- Namespace
  - `std::` specifies that we are going to use sth (i.e. `cout`) that belongs to "namespace" `std`
  - `std::` can be removed through use of `using` statements

### Simple C++ program (2)

```

1 // myFirst.cpp
2 #include <iostream>
3 using namespace std;
4
5 // function main
6 int main()
7 {
8     cout << "Welcome to C++!\n";
9
10    return 0;
11
12 } // end function main
    
```

Library file names in namespace `std` do not require a `.h` extension

We will be using objects located in special region/namespace called `std`.

Welcome to C++!

### Namespaces (1)

- Namespaces allow to group entities like classes, objects and functions under a single name.
- This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:
  - `namespace identifier`
    - {
    - entities
    - }

### Namespaces (2)

- Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:
 

```

namespace myNamespace
{
    int a, b;
}
            
```
- In order to access these variables from outside the `myNamespace` namespace we have to use the scope operator `::`

### Namespaces (3)

- The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:
 

```

#include <iostream>
using namespace std;
namespace first
{ int var = 5; }
namespace second
{ double var = 3.1416; }
int main ()
{
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
            
```

In this case, there are two global variables with the same name: `var`. One is defined within the namespace `first` and the other one in `second`. No redefinition errors happen thanks to namespaces.

5  
3.1416

### Namespaces (4) - using

The keyword `using` is used to introduce a name from a namespace into the current declarative region. E.g.:

```

#include <iostream>
using namespace std;
namespace first
{ int x = 5; int y = 10; }
namespace second
{ double x = 3.1416; double y = 2.7183; }
int main () {
    using first::x;
    using second::y;
    cout << x << endl;
    cout << y << endl;
    cout << first::y << endl;
    cout << second::x << endl; //...
}
    
```

Notice how in this code, `x` (without any name qualifier) refers to `first::x` whereas `y` refers to `second::y`, exactly as our using declarations have specified.

We still have access to `first::y` and `second::x` using their fully qualified names.

5  
2.7183  
10  
3.1416

### Namespaces (5) - using

The keyword `using` can also be used as a directive to introduce an entire namespace. E.g.:

```
#include <iostream>
using namespace std;
namespace first
{ int x = 5; int y = 10; }
namespace second
{ double x = 3.1416; double y = 2.7183; }
int main () {
    using namespace first;
    cout << x << endl;
    cout << y << endl;
    cout << second::x << endl;
    cout << second::y << endl;
    return 0;}
```

In this case, since we have declared that we were using namespace `first`, all direct uses of `x` and `y` without name qualifiers were referring to their declarations in namespace `first`.

5
10
3.1416
2.7183

31

### Namespaces (6) - using

`using` and `using namespace` have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. E.g., if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
#include <iostream>
using namespace std;
namespace first { int x = 5;}
namespace second { double x = 3.1416;}
int main () {
    { using namespace first;
      cout << x << endl;
    }
    { using namespace second;
      cout << x << endl;
    }
    //...
```

5
3.1416

32

### Namespaces (7)

- Namespace alias
  - We can declare alternate names for existing namespaces according to the following format:  
`namespace new_name = current_name;`
- `namespace std`

All the files in the C++ standard library declare all of its entities within the `std` namespace. That is why we have generally included the `using namespace std;` statement in all programs that used any entity defined in `iostream`.

33

### Using Namespaces

- Import namespace into scope
 

```
using namespace my;
string s; // implicitly use my::string
```
- Import individual symbols into scope
 

```
use my::string;
use your::vector;
string s; // my::string
vector v; // your::vector
```
- Explicitly qualify symbols
 

```
my::string s;
your::vector v;
```

34

### Structures and their applicability (1)

- Aggregate data types built using elements of other types
 

```
struct Time {
    int hour;
    int minute;
    int second;
};
```

Structure tag

Structure members
- Structure member naming
  - In same struct must have unique names
  - In different struct's can share name

35

### Structures and their applicability (2)

- Structure member can be pointer to instance of enclosing struct (self-referential structure) - Used for linked lists, queues, stacks and trees
- Structure member cannot be instance of enclosing the same structure
- Examples:
  - `Time timeObject;`
  - `Time timeArray[ 10 ];`
  - `Time *timePtr;`
  - `Time &timeRef = timeObject;`

36

### Structures and their applicability (3)

- Member access operators
  - Dot operator (.) for structure and class members
  - Arrow operator (->) for structure and class members via pointer to object
  - timePtr->hour  
same as (\*timePtr).hour
    - Parentheses required since \* has lower precedence than .

### Implementing a User-Defined Type **Time** with a **struct**

#### C-style structures

- No "interface"
  - If implementation of this data structure changes, all programs using that struct must change accordingly
- Cannot print as unit
  - Must print/format member by member
- Cannot compare in entirety
  - Must compare member by member

### Implementing **Time** Abstract Data Type with a **class**

- Classes
  - Model objects
    - Attributes (data members)
    - Behaviors (member functions)
  - Defined using keyword **class**
- Member access specifiers
  - public:**
    - Accessible wherever object of class in scope
  - private:**
    - Accessible only to member functions of class
  - protected:**

### Implementing **Time** Abstract Data Type with a **class**

- Constructor function
  - Special member function
    - Initializes data members
    - Same name as class
  - Called when object instantiated
  - You can have several constructors
    - Function overloading
  - No return type

### Class **Time** definition

```

1 class Time {
2
3 public:
4     Time(); // constructor
5     void setTime( int, int, int ); // set hr, min, sec
6     void printUniversal(); // in univ-time format
7     void printStandard(); // in strd-time format
8
9 private:
10    int hour; // 0 - 23 (24-hour clock format)
11    int minute; // 0 - 59
12    int second; // 0 - 59
13
14 }; // end class Time
    
```

### Class **Time** definition

```

1 class Time {
2
3 public:
4     Time(); // constructor
5     void setTime( int, int, int ); // set hr, min, sec
6     void printUniversal(); // in univ-time format
7     void printStandard(); // in strd-time format
8
9 private:
10    int hour; // 0 - 23 (24-hour clock format)
11    int minute; // 0 - 59
12    int second; // 0 - 59
13
14 }; // end class Time
    
```

## Implementing **Time** Abstract Data Type with a **class**

- **Objects of class**
  - **After class definition**
    - Class name can be treated as a new type specifier
      - ◆ **C++ extensible language**
    - Object, array, pointer and reference declarations
  - **Examples:**

Class name becomes new type specifier.

```

Time sunset; // object of type Time
Time arrayOfTimes[ 5 ]; // array of Time objects
Time *pointerToTime; // pointer to a Time object
Time &dinnerTime = sunset; // reference to Time object
                
```

43

## Implementing **Time** Abstract Data Type with a **class**

- **Member functions defined outside class**
  - **Binary scope resolution operator (::)**
    - “Ties” member name to class name
    - Uniquely identify functions of particular class
    - Different classes can have member functions with same name
  - **Format for defining member functions**

```

Return Type ClassName::MemberFunctionName ( ){
...
}
                
```
  - **Does not change whether function public or private**
- **Member functions defined inside class**
  - **Do not need neither a scope resolution operator, nor a class name**

44

```

2 // Time class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // Time abstract data type (ADT) definition
14 class Time {
15
16 public:
17     Time(); // constructor
18     void setTime( int, int, int ); // set hour, min, second
19     void printUniversal(); // print 24hrs-time format
20     void printStandard(); // print 12hrs-time format
21
22 private:
23     int hour; // 0 - 23 (24-hour clock format)
24     int minute; // 0 - 59
25     int second; // 0 - 59
26
27 }; // end class Time
                
```

Define class Time.

45

```

28
29 // Time constructor initializes each data member to zero and
30 // ensures all Time objects start in a consistent state
31 Time::Time()
32 {
33     hour = minute = second = 0;
34
35 } // end Time constructor
36
37 // set new Time value using universal time, perform validity
38 // checks on the data values and set invalid values to zero
39 void Time::setTime( int h, int m, int s )
40 {
41     hour = ( h >= 0 && h < 24 ) ? h : 0;
42     minute = ( m >= 0 && m < 60 ) ? m : 0;
43     second = ( s >= 0 && s < 60 ) ? s : 0;
44
45 } // end function setTime
46
47 ...
48 int main()
49 {
50     Time t; // instantiate object t of class Time
51
52 }
                
```

Constructor initializes private data members to 0.

public member function checks parameter values for validity before setting private data members.

Declare variable t to be object of class Time.

46

## Implementing **Time** Abstract Data Type with a **class**

- **Destructors**
  - **Same name as class**
    - Preceded with tilde (~)
  - **No arguments**
  - **Cannot be overloaded**
  - **Performs “termination housekeeping”**

47

## Implementing **Time** Abstract Data Type with a **class**

- **Advantages of using classes**
  - **Simplify programming**
  - **Interfaces**
    - Hide implementation
  - **Software reuse**
    - **Composition (aggregation)**
      - ◆ Class objects included as members of other classes
    - **Inheritance**
      - ◆ New classes derived from old

48

## Key features of C++

- **Classes, single / multiple inheritance**
- **Streamed I/O**
- **Exceptions**
- **Templates**
- **Namespaces**
- **Run-time type information**
- **Standard library**

49

## C++ vs. Java™

- |  |   |
|--|---|
| • <b>Hybrid</b>                                      | • <b>Everything is a class</b>                  |
| • <b>Real machine code</b>                           | • <b>Byte code</b>                              |
| • <b>Minimal standard libraries</b>                  | • <b>Massive APIs</b>                           |
| • <b>Memory management</b>                           | • <b>Garbage collection</b>                     |
| • <b>Single/Multiple inheritance</b>                 | • <b>Single inheritance</b>                     |
| • <b>Operator overloading</b>                        |   |
| • <b>Easy to access low level system information</b> | • <b>Difficult to access system information</b> |



50

*Questions?*

**Thank you very much for your attention**