Week 3: Introduction to C++

CSCI 2100 Data Structures
Fall 2017

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
swap using std::swap

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int a = 5, b = 3;
    cout << "a=" << a << " , b=" << b << endl;
    swap(a,b);
    cout << "a=" << a << " , b=" << b << endl;
    return 0;
}
```

### std::swap

Defined in header `<algorithm>`
Defined in header `<utility>`

```cpp
template< class T >
void swap( T& a, T& b );
```

(1) (until C++11)

```cpp
template< class T >
void swap( T& a, T& b ) noexcept(/* see below */);
```

(2) (since C++11)

```cpp
template< class T2, std::size_t N >
void swap( T2 (&a)[N], T2 (&b)[N] ) noexcept(/* see below */);
```

(2) (since C++11)
What version of C++ am I using now?

What version of g++ am I using now?
My g++ version check

```
[ahnt@hopper:~/Course/CSCI2100-Fall2017/Tests]$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-11)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

C++ Standards Support in GCC

GCC supports different dialects of C++, corresponding to the multiple published ISO standards. Which standard it implements can be selected using the \texttt{\texttt{-std=\}} command-line option.

- C++98
- C++11
- C++14
- C++1z
- Technical Specifications

For information about the status of the library implementation, please see this page.

C++1z Support in GCC

GCC has experimental support for the next revision of the C++ standard, which is expected to be published in 2017.

C++1z features are available as part of "mainline" GCC in the trunk of GCC's repository and in GCC 5 and later. To enable C++1z support, add the command-line parameter \texttt{-std=c++1z} to your \texttt{g++} command line. Or, to enable GNU extensions in addition to C++1z extensions, add \texttt{-std=gnu++1z}.

\textbf{Important}: Because the final ISO C++1z standard is still evolving, GCC's support is experimental. No attempt will be made to maintain backward compatibility with implementations of C++1z features that do not reflect the final standard.
How to simply check my GCC is C++11?

- `std::to_string` is working on `>= C++11`

```cpp
#include <iostream>
#include <string>

int main()
{
    double f = 23.43;
    std::string f_str = std::to_string(f);
    std::cout << "std::cout: " << f << std::endl;
    return 0;
}
```

```
$ g++ c11test_tostring.cpp -o c11test_tostring
c11test_tostring.cpp: In function ‘int main()’:
c11test_tostring.cpp:7:25: error: ‘to_string’ is not a member of ‘std’
  std::string f_str = std::to_string(f);
     ^
```
How to explicitly compile using C++11?

```
$ g++ -std=c++11 source.cpp -o output
```

- C++14 is not supported by our GCC version.

```
[ahnt@hopper:~/Course/CSCI2100-Fall2017/Tests]$ g++ -std=c++11 c11test_tostring.cpp -o c11test_tostring
[ahnt@hopper:~/Course/CSCI2100-Fall2017/Tests]$ ./c11test_tostring
std::cout: 23.43
```

- C++14 is not supported by our GCC version (4.8.X)
- You need to upgrade your g++ (GCC) at least to version 4.9.X
Be Careful! Algorithm header is unnecessary?

- Why can std::max and std::min still be used even if I didn't include <algorithm>?

```cpp
#include <iostream> // std::cout

int main () {
    std::cout << "min(1,2)=" << std::min(1,2) << std::endl;
    std::cout << "max(1,2)=" << std::max(1,2) << std::endl;

    int a = 5, b = 3;
    std::cout << "a= " << a << " , b= " << b << std::endl;
    std::swap(a,b);
    std::cout << "a= " << a << " , b= " << b << std::endl;

    return 0;
}
```

```
[ahnt@hopper:~/Course/CSCI2100-Fall2017/Tests]$ ./algorithm_header_test
min(1,2)==1
max(1,2)==2
a=5, b=3
a=3, b=5
```
Be Careful! Algorithm header is unnecessary?

- Most likely, something inside of iostream has directly or indirectly included some other header that defines std::min and std::max.

- Perhaps algorithm itself has been included. Perhaps some internal header that is used to implement your C++ standard library.

- You should not rely on this behavior. Include algorithm if you want std::min and std::max.

- However, include what is necessary as per the header's definition. But not more.
Function Arguments: Call By Value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

- In this case, changes made to the parameter inside the function have **no effect on the argument**.
Swap: Function Arguments: Call By Value

```c++
// function definition to swap the values.
void swap(int x, int y) {
    int temp;
    temp = x;     // save the value of x
    x = y;        // put y into x
    y = temp;     // put x into y
    return;
}

#include <iostream>
using namespace std;

int main () {
    int a = 100, b=200;  // local variable declaration:
    cout << "Before swap, values of a, b :" << a << ", " << b << endl;
    swap(a, b);  // calling a function to swap the values.
    cout << "After swap, values of a, b :" << a << ", " << b << endl;
    return 0;
}
```

Tests ahnt$ ./swap_value
Before swap, values of a, b :100, 200
After swap, values of a, b :100, 200
Function Arguments: Call By Reference

- The **call by reference** method of passing arguments to a function copies the **reference** of an argument into the formal parameter.

- Inside the function, the **reference** is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

- To pass the value by reference, argument reference is passed to the functions just like any other value.
Swap: Function Arguments: Call By Reference

```cpp
// function definition to swap the values.
void swap(int &x, int &y) {
    int temp;
    temp = x; // save the value of x
    x = y; // put y into x
    y = temp; // put x into y
    return;
}

#include <iostream>
using namespace std;

int main () {
    int a = 100, b=200; // local variable declaration:
    cout << "Before swap, values of a, b :" << a << ", " << b << endl;
    swap(a, b); // calling a function to swap the values.
    cout << "After swap, values of a, b :" << a << ", " << b << endl;
    return 0;
}
```

CAS-MCS-7Q22G4:Tests ahnt$ ./swap_reference
Before swap, values of a, b :100, 200
After swap, values of a, b :200, 100
CAS-MCS-7Q22G4:Tests ahnt$
Why pass by reference?

- Changes persist outside of function
- Saves time and space
- You don’t have to pass the address of a variable
- You don’t have to dereference the variable inside the called function.
Function Arguments: Call By Pointer

- The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter.
- Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- To pass the value by pointer, argument pointers are passed to the functions just like any other value.
Swap: Function Arguments: Call By Pointer

```
// function definition to swap the values.
void swap(int *x, int *y) {
    int temp:
    temp = *x;    // save the value at address x
    *x = *y;      // put y into x
    *y = temp;    // put x into y

    return;
}

#include <iostream>
using namespace std;

int main () {
    int a = 100, b=200;    // local variable declaration:
    cout << "Before swap, values of a, b :" << a << ", " << b << endl;
    swap(&a, &b);    // indicates pointers to a and b (addresses)
    cout << "After swap, values of a, b :" << a << ", " << b << endl;
    return 0;
}
```

CAS-MCS-7Q22G4:Tests ahnt$ ./swap_reference
Before swap, values of a, b :100, 200
After swap, values of a, b :200, 100
CAS-MCS-7Q22G4:Tests ahnt$
*: dereference operator

```cpp
int x;
x = 12;

int* ptr;
ptr = &x;

cout << *ptr;
```

**NOTE:** The value pointed to by `ptr` is denoted by `*ptr`
How to call/access pointer variables?

Option 1: `(*variable).method`

Option 2: `variable->method`
Dynamic Memory Management

```
Point *p;
// declare pointer variable (not yet initialized)
p = new Point(); // dynamically allocate a new Point instance, storing its address
```

- When a declaration is made, the system reserves memory for storing the state of the object.
- There are circumstances when a programmer wants to take a more active role in controlling the underlying memory management.
- In C++, this is accomplished using the `new` and `delete` operators
  - `new` is used to allocate memory during execution time
    - returns a pointer to the address where the object is to be stored
    - always returns a pointer to the type that follows the `new`
In C and C++, three types of memory are used by programs:

- **Static memory** - where global and static variables live

- **Heap memory** - dynamically allocated at execution time
  - "managed" memory accessed using pointers

- **Stack memory** - used by automatic variables

<table>
<thead>
<tr>
<th>Static Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Variables</td>
</tr>
<tr>
<td>Static Variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap Memory (or free store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically Allocated Memory</td>
</tr>
<tr>
<td>(Unnamed variables)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Variables</td>
</tr>
<tr>
<td>Function parameters</td>
</tr>
</tbody>
</table>
3 Kinds of Program Data

- **STATIC DATA**: Allocated at compiler time

- **DYNAMIC DATA**: Explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators `new` and `delete`

- **AUTOMATIC DATA**: Automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function
The **NULL** Pointer

- There is a pointer constant called the “null pointer” denoted by **NULL**
- But **NULL** is not memory address 0.

**NOTE:** It is an error to dereference a pointer whose value is **NULL**. Such an error may cause your program to crash, or behave erratically. It is the programmer’s job to check for this.

```c
while (ptr != NULL) {
    . . . // ok to use *ptr here
}
```
Operator **delete**

```cpp
char* ptr;

ptr = new char;
*ptr = 'B';

NOTE:
**delete** deallocates the memory pointed to by ptr

cout << *ptr;

delete ptr;
```
Class Member Function

- A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
- It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

```java
class Box {
    public;
    double length;    // length of a box
    double breadth;   // breadth of a box
    double height;    // height of a box

    double getVolume(void) {
        return length*breadth*height;
    }
};
```
Class Member Function

- You can define the same function outside the class using the scope resolution operator (::) as follows

```cpp
class Box {
    public:
        double length;  // Length of a box
        double breadth;  // Breadth of a box
        double height;  // Height of a box

    // Member functions declaration
    double getVolume(void);
};

// Member functions definitions
double Box::getVolume(void) {
    return length * breadth * height;
}
```
```
#include <iostream>

using namespace std;

class Box {
    public:
        double length;    // Length of a box
        double breadth;   // Breadth of a box
        double height;    // Height of a box

        // Member functions declaration
        double getVolume(void);
    };

    // Member functions definitions
    double Box::getVolume(void) {
        return length * breadth * height;
    }

    // Main function for the program
    int main() {
        Box Box1;           // Declare Box1 of type Box
        double volume = 0.0;

        Box1.height = 5.0;
        Box1.length = 6.0;
        Box1.breadth = 7.0;

        // volume of box 1
        volume = Box1.getVolume();
        cout << "Volume of Box1 : " << volume << endl;

        return 0;
    }
```
```cpp
#include <iostream>
using namespace std;

class Box {
    public:
        double length;
        void setWidth(double wid);
        double getWidth(void);
    
    private:
        double width;
};

// Member functions definitions
double Box::getWidth(void) {
    return width;
}

void Box::setWidth(double wid) {
    width = wid;
}

// Main function for the program
int main() {
    Box box;
    // set box length without member function
    box.length = 10.0; // OK: because length is public
    cout << "Length of box : " << box.length << endl;

    // set box width without member function
    // box.width = 10.0; // Error: because width is private
    box.setWidth(10.0); // Use member function to set it.
    cout << "Width of box : " << box.getWidth() << endl;
    return 0;
}
```
The Class Constructor

- A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
- A constructor will have exact same name as the class and it does not have any return type at all, not even void.
- Constructors can be very useful for setting initial values for certain member variables.
```cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        // Constructor initialize the values with args
        Rectangle (int a, int b) {
            width = a;
            height = b;
        }

        int area () {
            return (width*height);
        }

    private:
        int width, height;
};

// Main function for the program
int main() {
    Rectangle rect1 (3,4);
    Rectangle rect2 (5,6);
    cout << "Area of rect1 : " << rect1.area() << endl;
    cout << "Area of rect2 : " << rect2.area() << endl;
    return 0;
}
```
```cpp
#include <iostream>

using namespace std;

class Rectangle {
   public:
      Rectangle ();
      Rectangle (int, int);
      int area () {
         return (width*height);
      }

   private:
      int width, height;
};

Rectangle::Rectangle () {
   width = 5;
   height = 5;
}

Rectangle::Rectangle (int a, int b) {
   width = a;
   height = b;
}

int main() {
   Rectangle rect1;
   Rectangle rect2 (3, 4);
   cout << "Area of rect1 : " << rect1.area() << endl;
   cout << "Area of rect2 : " << rect2.area() << endl;
   return 0;
}
```
```cpp
#include <iostream>

using namespace std;

class Rectangle {
  public:
    Rectangle (int, int);
    int area () {
      return (width*height);
    }
  
  private:
    int width, height;
};

// constructor can be defined using member initialization
Rectangle::Rectangle (int a, int b): width(a), height(b) {}

int main() {
  Rectangle rect1 (5, 5);
  Rectangle rect2 (3, 4);
  cout << "Area of rect1 : " << rect1.area() <<endl;
  cout << "Area of rect2 : " << rect2.area() <<endl;
  return 0;
}
```
```cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        Rectangle (int a, int b): width(a), height(b) {}
        int area () {
            return (width*height);
        }
    
    private:
        int width, height;
};

int main() {
    Rectangle rect1 (5, 5);
    Rectangle * rect2;
    rect2 = new Rectangle (3, 4);
    cout << "Area of rect1 : " << rect1.area() << endl;
    //cout << "Area of rect2 : " << rect2.area() << endl;
    cout << "Area of rect2 : " << (*rect2).area() << endl;
    cout << "Area of rect2 : " << rect2->area() << endl;
    return 0;
```
Class Destructors

- Destructors (opposite of constructors) are functions which destroy the object whenever the object goes out of scope.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
- It has the same name as that of the class with a tilde (~) sign before it.

```cpp
class A {
  public:
    ~A();
};
```
```cpp
#include <iostream>

using namespace std;

class Rectangle {
   public:
      Rectangle (int a, int b): width(a), height(b) {
         cout << "Object is being created" << endl;
      }
      ~Rectangle() {
         cout << "Object is being delete" << endl;
      }
      int area () {
         return (width*height);
      }

   private:
      int width, height;
};

int main() {
   Rectangle rect1 (5, 5);
   cout << "Area of rect1 : " << rect1.area() << endl;

   Rectangle * rect2;
   rect2 = new Rectangle (3, 4);
   cout << "Area of rect2 : " << (*rect2).area() << endl;
   cout << "Area of rect2 : " << rect2->area() << endl;
   return 0;
}
```
Class Destructors

Notes

● The destructors will get automatically called even if we do not explicitly call them.

● However, if you allocated memory at run time using `new` operator, then you should use delete operator to de-allocate the memory.
int main() {
    Rectangle rect1 (5, 5);
    cout << "Area of rect1 : " << rect1.area() << endl;

    Rectangle * rect2;
    rect2 = new Rectangle (3, 4);
    cout << "Area of rect2 : " << (*rect2).area() << endl;
    cout << "Area of rect2 : " << rect2->area() << endl;
    delete rect2;

    return 0;
}
Compiling and Linking

```
$ g++ -o main.exe main.cpp

$ g++ -c main.cpp
$ g++ -o main.exe main.o
```
Separating class code into a header and cpp file

$ g++ -o main.exe add.cpp main.cpp
GCD project with three files

```cpp
#include "gcd.h"

int gcd(int u, int v) {
    /* We will use Euclid's algorithm for computing the GCD */
    int r;
    while (v != 0) {
        r = u % v;  // compute remainder
        u = v;
        v = r;
    }
    return u;
}
```

```cpp
#include "gcd.h"

int main() {
    int a, b;
    cout << "First value: ";
    cin >> a;
    cout << "Second value: ";
    cin >> b;
    cout << "gcd: " << gcd(a, b) << endl;
    return 0;
}
```
```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

using namespace std;

class Rectangle {
    public:
        Rectangle (int a=0, int b=0);
        ~Rectangle();
        int area ();

    private:
        int width, height;
};

#endif
```
```cpp
1 #include <iostream>
2 #include "rectangle.h"

3 using namespace std;

4 Rectangle::Rectangle (int a, int b): width(a), height(b) {
5     cout << "Object is being created" << endl;
6 }

7 Rectangle::~Rectangle() {
8     cout << "Object is being delete" << endl;
9 }

10 int Rectangle::area () {
11     return (width*height);
12 }
```
rectangle_test.cpp

```cpp
#include <iostream>
#include "rectangle.h"
using namespace std;

int main() {
    Rectangle rect1 (5, 5);
    cout << "Area of rect1 : " << rect1.area() << endl;
    Rectangle * rect2;
    rect2 = new Rectangle (3, 4);
    cout << "Area of rect2 : " << (*rect2).area() << endl;
    cout << "Area of rect2 : " << rect2->area() << endl;
    delete rect2;
    return 0;
}
```
Finally, we note that for larger problems, there are other tools to assist developers in managing their projects. For example, many Integrated Development Environments (IDEs) will keep track of which pieces of source code have been modified, and which need to be re-compiled when building a new executable.

One of the classic tools for developers in managing the (re)building of a project is a program known as **make**. This program relies upon a configuration file for the project, conventionally named **makefile**. The **makefile** designates what components comprise the project, and upon which pieces of source code each component depends. The make command causes a re-build of the entire project, but relying on the file-system timestamps to determine which pieces of source code have been edited since the previous build.
Object demo and makefile example

hopper:/public/ahnt/courses/csci2100/demos/objectdemo