Dalhousie University

Faculty of Computer Science

CSCI 3132 – Object Orientation and Generic Programming

Week 7 – References

What are C++ References

- Another name for a variable
  - Can be used to read/modify the original data stored in that variable
- Why not use the original variable?
  - Reference can access the original variable even if it is located in a different scope
  - Possible to make the function arguments references and you get a way to change the original data passed into the function.
    - Different from arguments to a function copied into new variables

Reference versus Pointer

- How is a reference different from a pointer?
  - A reference must be initialized with another variable when it is created
  - A reference cannot be re-assigned, unlike a pointer
  - A reference always points to an object while a pointer can be NULL (i.e. point nowhere)
  - You can't take the address of a reference like you can with pointers
  - No reference arithmetics
    - However, you can take the address of an object pointed by a reference and do pointer arithmetics on it (e.g. &x + 1)

References

- What does the following code do?

```
int x;
int &ref = x;  //declares a reference to int x
ref = 10;      //changes the value of x to 10
cout<<ref<<endl;
```

- References as function arguments – what is the output for:

```
void square(int &x){
    x *= x;
}
int main(){
    int x = 10;
    int& ref = x;
    square(ref);
    std::cout<<x<<std::endl;    //output is 100
}
```

const References

- Use const references when you don't want the variable passed as an argument to be changed
  - E.g. `int func(const foo &obj1){ }`
  - You can work with objects instead of pointers
  - This ensures that the object is not modified
  - Advantage is that copy of object is not created
  - Advantage over pointer is that the code is clearer
    - No need to create a separate pointer variable and pass it as an argument

Example

```cpp
#include<iostream>
using namespace std;
void Test1(int &x, int &y){
    cout << "Enter two numbers" << endl;
    cin >> x >> y;
}
void Test2(int &x, int &y, int &z){
    x *= 2;
    y *= 2;
    z *= 2;
}
void Test3(int x, int y, int z){
    x /= 2;
    y /= 2;
    z /= 2;
}
void Display(const int &x, const int &y, const int &z){
    cout << x << ", " << y << ", " << z << endl;
}
```

```cpp
int main() {
    int a = 1, b = 1, c = 1;
    Display(a, b, c);
    Test1(a, b);
    Display(a, b, c);
    Test2(a, b, c);
    Display(a, b, c);
    Test3(a, b, c);
    Display(a, b, c);
}
```

```
1, 1, 1
Enter two numbers
3
4
3, 4, 1
6, 8, 2
6, 8, 2
```

Exercise

- Write a function that takes two strings as input and swaps them using (i) string objects, (ii) pointers, and (iii) references

```
void swap (______ a, ______ b){
      ...
}
void Display (______ a, ______ b){
      cout<<a<<b<<endl;
}
int main(){
      __________ = " Hello ";
      __________ = " World ";
      Display(_______, _______);
      Swap(_______, _______);
      Display(_______, _______);
}
```

Output should be:

```
Hello  World
World  Hello
```

Example – Solution

Using references – Better

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

using namespace std;

void Swap(string& a, string &b){
    string tmpstr = a;
    a = b;
    b = tmpstr;
}
void Display(const string & a, const string& b){
    cout<<a<<b<<endl;
}
int main(){
    string str1 = " Hello ";
    string str2 = " World ";
    Display(str1, str2);
    Swap(str1, str2);
    Display(str1, str2);
}
```

Using pointers – Works

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

using namespace std;

void Swap(string* a, string *b){
    string tmpstr = *a;
    *a = *b;
    *b = tmpstr;
}
void Display(string * a, string* b){
    cout<<*a<<*b<<endl;
}
int main(){
    string str1 = " Hello ";
    string str2 = " World ";
    Display(&str1, &str2);
    Swap(&str1, &str2);
    Display(&str1, &str2);
}
```

When to use what?

- Use pointers if:
  - Its possible that there might be nothing to refer to
    - Set pointer to NULL
  - You want to refer to different things at different times
  - You want to be obvious what's going on
- References are safer
  - Use references if you can, pointers if you must!
- Use references:
  - If you know there will always be an object to refer to
  - As function arguments
- When returning a reference
  - The object being referred should not go out of scope.
  - Not legal to return a reference to local var.
  - A reference can be returned on a static variable.

Constructors

Constructor

- A member function that is automatically called when a class object is created
  - No return type
  - Used to initialize an object's attributes
- Good practice to always write a constructor for every class
- Can be overloaded
- Syntax:
  - *Classname::Classname(parameters)*

Types of Constructors

- Default constructor
  - Takes no arguments
  - Compiler writes an empty one if you don't
- Constructor with default arguments
  - Default arguments used when not passed in the call
  - Is a default constructor when all arguments have a default value
- Overloaded constructor
  - Can have one or more arguments
- Copy constructor
  - Special constructor that is called whenever a new object is created and initialized with another object's data
  - Required when the default member-wise assignment cannot be used

Example

```
class Point {
        double ptX, ptY;
public:
   Point() { ptX = 0; ptY = 0; }    //default constructor
   Point(double x = 0, double y = 0) : ptX(x), ptY(y) {} //error & ambiguous
   Point(double x, double y = 0) : ptX(x), ptY(y) {}     //overloaded
   Point(double x, double y) : ptX(x), ptY(y) {}         //error
   void Display() { cout << ptX << ", " << ptY << endl; }
};
int main() {
   Point pt1;
   Point pt2(5);
   Point pt3(7, 7);
   pt1.Display();    //0, 0
   pt2.Display();    //5, 0
   pt3.Display();    //7, 7
   pt3 = pt2;        //member-wise assignment
   pt3.Display();    //5, 0
}
```

Problem with Member-wise Assignment

- Consider the following member-wise assignment:
  - StudentGrades stu_grade2 = stu_grade1;
  - stu_grade2's constructor not called.
    - Member-wise assignment will copy each of the stu_grade1 object's variables to stu_grade2 object's variables
- What happens when one of the attributes is a pointer to an array?
  - In above example, the object is initialized with another object
    - Separate section of memory is not allocated for the array to which that pointer points – both pointers point to the same array
  - What if one of the objects is destroyed?
    - Destructor called that frees the memory and pointer of the other object still points to that place in memory
- Copy Constructor solves the problem

Copy Constructor

- Copy constructor is called when an object is initialized with another object's data
  - Same form as other constructors, but it has a reference parameter of the same class type as the object itself
- Example:
  - ```
    StudentGrades (StudentGrades & sg){
       st_name = sg.st_name;
       st_num_subs = sg.st_num_subs;
       st_scores = new int[st_num_subs];
       for (int i=0; i<st_num_subs; i++)
            st_scores[i] = sg.st_scores[i];
    }
    ```
  - Now we can do this:
    ```
    StudentGrades stu_grade1 = ("Khurram", 5);
    StudentGrades stu_grade2 = stu_grade1;
    ```
    - This will call the copy constructor with stu_grade1 object passed as a reference to it.
    ```

Copy Constructor

```
class Point {
        double ptX, ptY;
public:
        Point() { ptX = 0; ptY = 0; }
        Point(double x, double y = 0) : ptX(x), ptY(y) {}
        Point(Point & pt) { ptX = pt.ptX; ptY = pt.ptY; }
        void Display() { cout << ptX << ", " << ptY << endl; }
};
int main() {
        Point pt1;
        pt1.Display();
        Point pt2(5);
        pt2.Display();
        Point pt3 = pt2;
        pt3.Display();
}
```

Generic Programming

What is Generic Programming

- *Programming paradigm for developing efficient, reusable software that
  - Focuses on finding commonality among similar implementations of the same algorithm
  - Provides suitable abstractions so that a single, generic algorithm can cover many concrete implementations.

*Ref: generic-programming.org

What is Generic Programming

- Writing "templates" of source codes
  - Parameters unspecified at the time of class definition
  - Programs evaluated at compile time
  - Instantiation of the templates provide information to the compiler for the type it should use to create the class out of the template

Template – Example

- Templates can be:
  - Function templates that behave like functions that can accepts many different kinds of arguments
  - Example:

```
 template <class T>
T max(T x, T y){
      if (x < y)   return y;
      else         return x;
}
```

    - Compare this with
      `#define max(a,b)   ((a) < (b) ? (b) : (a))`
  - Class templates that are often used to make generic containers
  - Example: Linked list container of the STL library
    - list<*type*> has a set of standard functions associated with it that work, regardless of what the type is.
    - *more on these later*

Templates

- Advantages:
  - Already looked at while studying templates
  - Type-safety, code reusability, …
- Disadvantages:
  - Most compilers have poor support for templates
  - Difficult to make sense of error messages
    - Executed code is generated by the compiler and is not present in the original code
  - Compiler generates extra code for each use of template (template instantiation)
    - Indiscriminate use may result in code bloat and large executables
    - Separate compilation of template definitions and template function declarations not implemented by most compilers

Examples of Generic Programming

- Standard Template Libraries (STL)
  - Generic C++ library of container classes, algorithms and iterators
  - Almost every component in the STL is a template
    - Can be instantiated to contain any type of object
  - STL algorithms are decoupled from the STL container class
    - Usually implemented as global functions
  - STL iterators are generalization of pointers
    - Iterators make it possible to decouple algorithms from containers

Concepts

- Consider the following piece of code:

```
template <class InputIterator, class T>
InputIterator find      (InputIterator first,
                        InputIterator last,
                        const T& value) {
      while (first != last && *first != value)
            ++first;
      return first;
}
```

  - What is the set of types that can be correctly substituted for the formal template parameters
    - int *, double * but not int or double
  - Find(…) implicitly defines a set of requirements on types

Concepts

- Types substituted in find must be able to provide certain operations, for example:
  - Compare two objects (first != last) of the type
  - Increment an object of that type (++first)
  - Dereference an object of that type to obtain an object that it points to (*first != value)
- Such a set of type requirements is called a concept
  - A type conforms to a concept if it satisfies all of those requirements for that concept

Concepts

- Using concepts makes it possible to write programs that separate interface from implementation
  - While creating the find template function, the author only has to consider the interface provided by the concept (in this case input iterator)
    - No worries about implementation of every possible type that conforms to that concept
  - While using the find function, one only needs to ensure that the passed arguments are models of the input iterator
- Programming in terms of concepts rather than specific types makes it possible to reuse and combine software components together

STL Container Classes

- STL includes the following classes:
  - vector
  - list
  - deque
  - set
  - multiset
  - map
  - multimap
  - hash_set, hash_multiset, hash_map and hash_multimap