References, Argument Passing, and Constants

Chapter 6

Objectives

- **Explain the call-by-value mechanism and the implicit invocation of constructors in passing objects as arguments.**
- **Use reference declarations to alias variables.**
- **Use references in argument passing.**
- **Explain the role of copy constructors.**
- **Use constant types in your programs.**

  - The example programs are in the **chapter directory.**
  - Labs located in Labs/Lab6

Variables

- **Both a symbolic variable and a literal constant maintain storage and have an associated type.**
- **A symbolic variable is addressable. There are two values associated with a variable:**
  - Its data value or r-value, stored at some location in memory
  - Its location value or l-value, which is the address in memory at which the data value is stored
- **In an assignment the left hand side is an l-value and the right hand side is an r-value.**

Argument Passing

- **Functions are allocated storage on the run-time stack.**
  - This storage area is known as the activation record. It is popped when function is no longer active.
- **The formal arguments of a function are provided storage in the activation record.**
- **The actual arguments of a function are the expressions between commas in the argument list of the function call.**

Call-by-Value

- **Argument passing is the process of initializing the storage of the formal arguments by the actual arguments.**
- **Default method of argument passing in C++ is call-by-value, in which the r-values of the actual arguments are copied into the storage of the formal arguments.**
- **Call-by-value is "safe": the function never directly accesses the actual arguments, only its own local copies.**
- **There are drawbacks to call-by-value:**
  - Overhead in copying a large object
  - When it is desired to modify the value of an argument, resort must be made to pointers

Reference Declarations

- **Reference declarations are of the form:**
  - type& identifer = object
- **The identifier is an alternative name or alias for the object.**
- **Lexically, the ampersand & can be right after the type, just before the identifier, or separated by space from both.**

Reference Declarations (continued)

```
int i = 3;
Int &j = i;              // j is an alias for i
Int &k = i;              // another alias
. . .
i = 5;                   // now j (and k and m) will also contain 5
const char &tab = '\t';  // can initialize const alias to literal
```

Call-by-Reference

- **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. *Changes made to the parameter affect the passed argument*.**
- **The semantics of argument passing are identical to the semantics of initialization.**

```
void swap(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main(){
    int x = 7, y = 5;
    swap(x, y);
}
```

Demo

- **Open the folder ReferenceDemo and examine the code.**
- **Run the program and examine the result.**

Copy Constructor

- **What happens when an object is passed by value?**
  - Just as for an ordinary data item, a copy must be made
  - For an object from a user defined class a constructor must be invoked, called the **copy constructor**

```
class String{
public:
    . . .
    String(const String& s); // by reference
    . . .
};
```

Copy Constructor

- **IMPORTANT - Notice that the argument to the copy constructor must be passed by reference.**
- If it were passed by value, copy constructor would have to be called within the copy constructor, leading to an infinite regress!

Default Copy Constructor

- **If you do not declare a copy constructor, the compiler will create one for you.**
- **This *default* copy constructor will initialize each data member of the class by copying its counterpart to the original.**
- **This default copy constructor is adequate for a class (or struct) like *Date*, where the entire state of the object is stored within the object:**

Revisit Our String Class

- The folder StringBug contains the current version of the *String* class that does not implement a copy constructor.
- Review and run the program, you will see the program does not work because of the default copy constructor!
- Why? The pointer is copied, leaving two String objects pointing at the same memory.

Demo

- StringCopy contains a new version of the *String* class that implements a copy constructor.
- Examine the copy constructor prototype and implementation.
- Run the program.

Review of Constant Types

- ***const* type modifier turns a symbolic variable into a *symbolic constant*.**
  - const int stacksize = 100;
- **A symbolic constant is like a variable in having a memory location and a type, but is *read only*:**
  - stacksize = 150;   // illegal
- **A symbolic constant *must* be initialized when it is declared.**
  - const int stacksize;   // illegal
- **You cannot assign the address of a symbolic constant to a pointer. (Otherwise, the value of the constant could get changed indirectly through the pointer.)**
  - int *p = &stacksize;   // illegal
- **const names can be inspected in a debugger.**

Constant and Pointers

- Both pointer and what’s pointed to can be const

```
char greeting[] = "Hello";

// non-const pointer, non-const data
char* p = greeting;
// non-const pointer, const data
const char* q = greeting;
// const pointer, non-const data
char* const r = greeting;
// const pointer, const data
const char* const s = greeting;
```

Constants and Arguments

- **Whenever you do not intend for a reference argument to be modified from within a function, you should declare the argument as *const*.**

```
struct Table
{
   int   data[100];
   char  name[20];
};
int search(const Table& t, int target)
{
    . . .
    t.data[0] = 5;  // illegal, caught by compiler
    . . .
}
```

Constants and Arguments (continued)

- Similarly for pointer arguments:

```
int search(const Table* pt, int target){
   . . .
   pt->data[0] = 5;  // illegal, caught by compiler
   . . .
}
```

Chains of Function Calls

- **The call to *lookup* violates the *const* declaration of the argument of *search*.**

```
struct Table{
   int  data[100];
   char name[20];
};
extern int lookup(Table&, int);
int search(const Table& t, int target){
   return lookup(t, target);      // compile error!!
   . . .
}
```

const Objects

- **The *const* keyword can also be used when declaring a user defined object:**
  - const Complex i(0, 1); // i cannot be changed
- **Compiler will now prevent access to *all* member functions for *i*:**
  - i.setReal(2); // illegal
  - float a = i.getReal(); // illegal

const Member Functions

- **Constant member functions can not change the values of the data members of their class.**
- **To make a member function constant, append *const* to the function prototype and also to the function definition header.**
- **Non-const functions can only be called by non-const objects.**

```
class Complex{
public:
    Complex (float real, float imag);
    void setReal(float x);
    float getReal() const;        // read-only
    void setImaginary(float x);
    float getImaginary() const;   // read-only
};
```

Demo

- **Review and run the application in folder Const which demonstrates constant objects and constant member functions..**

Summary

- **Default argument passing in C++ is call-by-value, which involves copying of data from caller to called function.**
- **When an object of a user-defined class is passed, a constructor is implicitly invoked to do the copy.**
- **You should implement a copy constructor for a class where the object stores a pointer to heap data.**
- **Reference declarations can be used to make an alias for variables.**
- **References can be used for more efficient and intuitive argument passing.**
- **Constant types can be used to protect against modification of data that should not be changed.**