

C/C++ Programming Language

CS219 Spring

Feng Zheng

Lecture 11



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



Content

- Review
- An example of string class: problems
- Compiler **automatically** generates functions
 - Default constructors
 - Copy constructors
 - Assignment operators
- **Improved** string class
 - Comparison, accessing characters
- **Pointers** to objects

Brief Review



Review

- Operator Overloading
 - Operator function
 - Friends
 - Example: overloading the << operator
- Automatic Conversions and Type Casts for Classes (=)
 - Type cast from **single** augment to an object (constructor)
 - ✓ Implicit constructor
 - ✓ Explicit constructor
 - Conversion function (**operator**)



Dynamic Memory and Classes



The Reasons for Dynamic Memory

- Problems: some things were **not confirmed** during programming
 - What would you like for breakfast, lunch, and dinner for the next month?
 - How many ounces of milk for dinner on the 3rd day?
 - How many raisins in your cereal for breakfast on the 15th day?
- Letting the program **decide about memory** during **runtime** rather than during compile time
 - Memory use can depend on the **needs of a program** instead of on a rigid set of storage-class rules
 - C++ utilizes the **new** and **delete** operators
 - ✓ **Destructors** can become necessary
 - ✓ Have to overload an **assignment operator** to get a program to behave properly



A Review Example and Static Class Members

- Run `vegnews.cpp`, `strbad.h`, `strbad.cpp`
- Problems
 - Passing an object as a function argument somehow causes the **destructor to be called**
 - Although passing by value is supposed to protect the original argument from change, the function **messes up the original string** beyond recognition, and some nonstandard characters get displayed
 - The number of **constructor** calls does **not equal** the number of **destructor** calls
- Compiler **automatically** generates the constructor

```
StringBad sailor = sports;
```

```
StringBad sailor = StringBad(sports); //constructor using sports
```



Special Member Functions

- C++ automatically provides the following member functions
 - A **default constructor** if you define no constructors
 - A **copy constructor** if you don't define one
 - An **assignment operator** if you don't define one
 - A **default destructor** if you don't define one
 - An **address operator** if you don't define one



Default Constructors

- C++ provides you with a **default** constructor, if you **fail** to provide any constructors at all

```
Klunk::Klunk() { } // implicit default constructor
```

- Define a **default** constructor explicitly
 - **No** arguments
 - **Or** all its arguments have **default values**
- Can have **only one** default constructor

```
Klunk() { klunk_ct = 0 } // constructor #1  
Klunk(int n = 0) { klunk_ct = n; } // ambiguous constructor #2
```

```
Klunk kar(10); // clearly matches Klunk(int n)  
Klunk bus; // could match either constructor
```



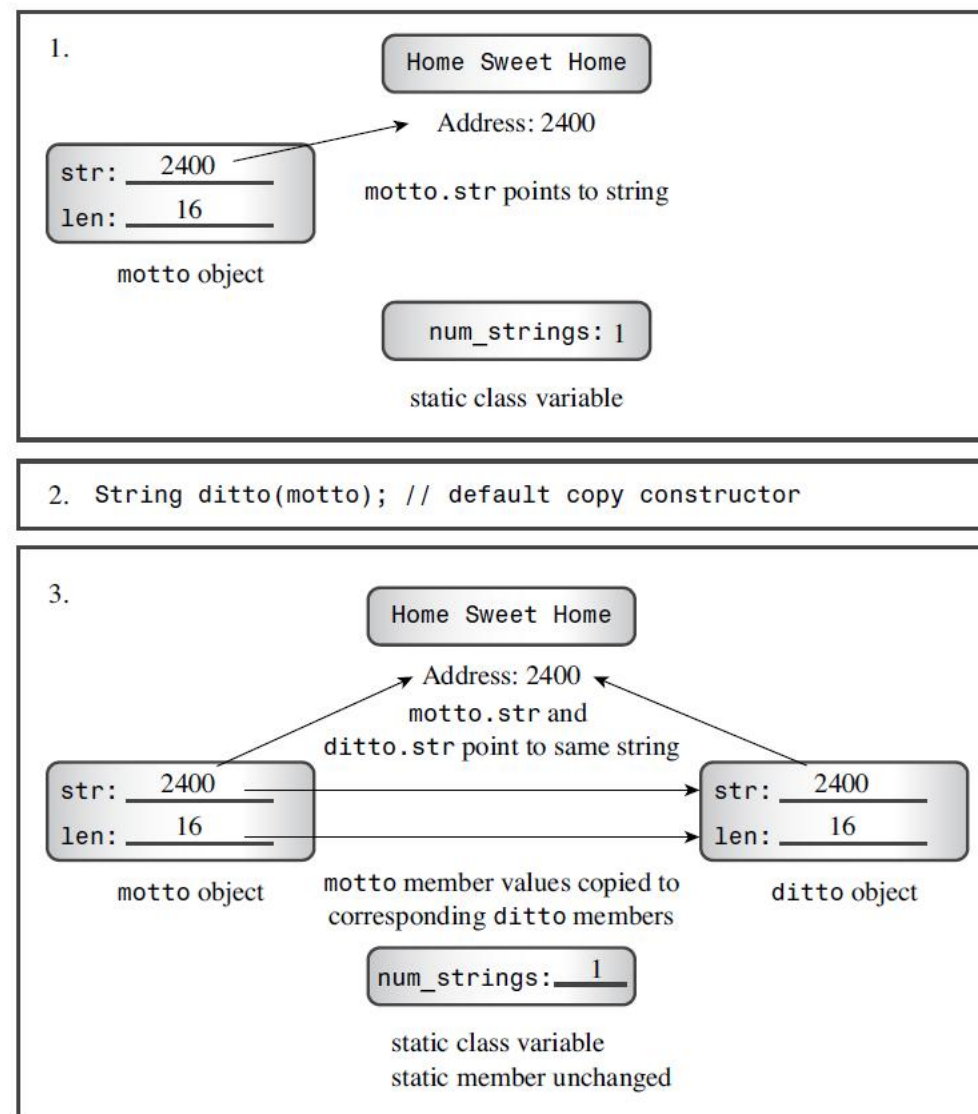
Copy Constructors

- A **copy** constructor is used to copy an object to a **newly** created object
- When a copy constructor is used
 - A copy constructor is invoked whenever a **new** object is **created** and **initialized** to an **existing** object of the same kind
 - A compiler also uses a copy constructor whenever it generates **temporary** objects (passing value to functions)
- What a **default** copy constructor does
 - Perform a **member-by-member copy** of the **nonstatic** members
 - Static members are unaffected because they belong to the **class as a whole** instead of to individual objects



Default Copy Constructors

- An inside look at **member-wise** copying
- Two problems
 - Point to the **same** address
 - Static variables **unchanged**





Back to Stringbad: Where the Copy Constructor Goes Wrong

- First problem: two more objects destroyed than constructed
 - Default copy constructor **doesn't increment** the counter. However, the destructor **does update** the counter
 - The solution is to provide an **explicit** copy constructor
- Second problem: default copy constructor **does not copy** the string; it copies the pointer to a string
 - The same memory location that has already been freed by the destructor

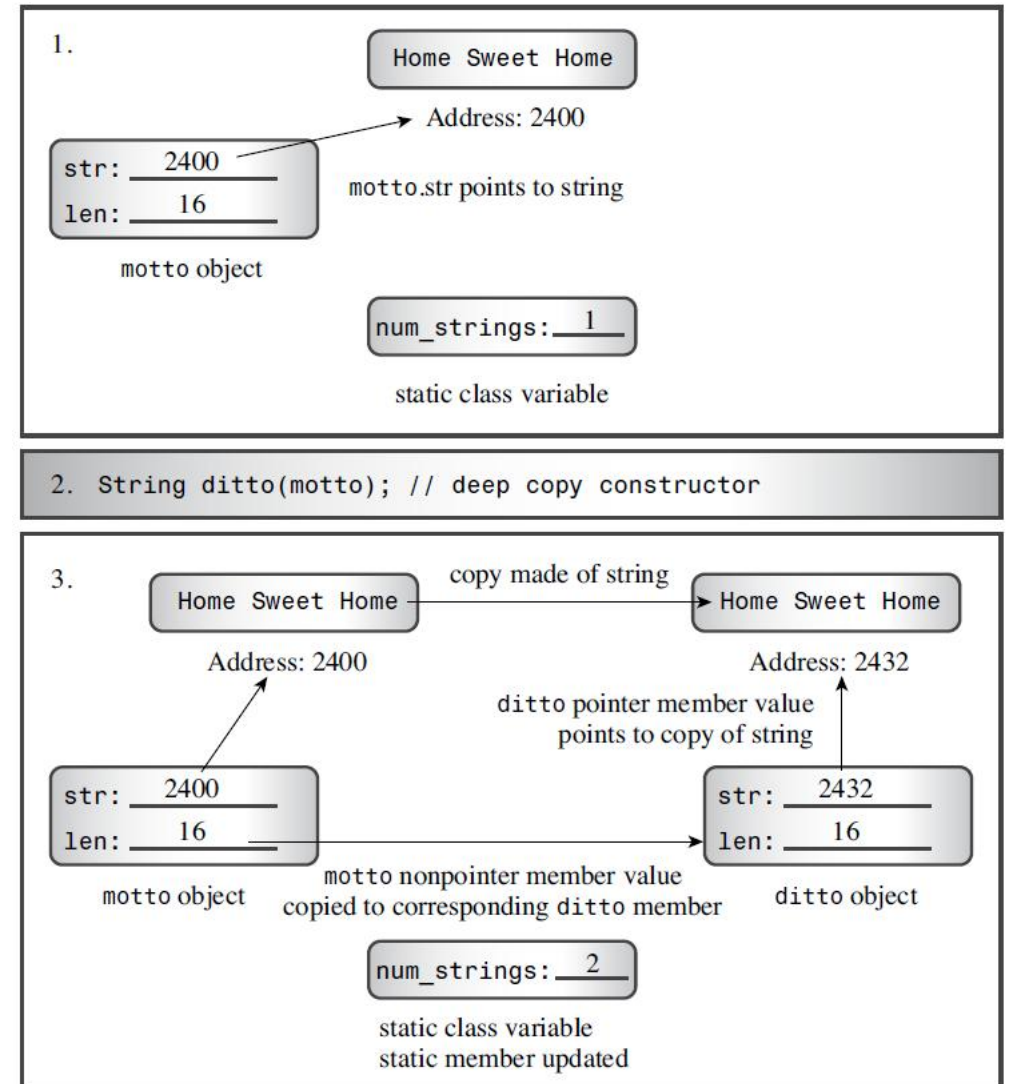


Fixing the Problem by Defining an Explicit Copy Constructor

- The **explicit** copy constructor
 - **Duplicate** the string
 - **Assign** the address of the duplicate to the member

```
StringBad::StringBad(const StringBad & st)
{
    num_strings++;           // handle static member update
    len = st.len;           // same length
    str = new char [len + 1]; // allot space
    std::strcpy(str, st.str); // copy string to new location
    cout << num_strings << ": \"" << str
         << "\" object created\n"; // For Your Information
}
```

- Uncomment copy constructor





More Stringbad Problems: Assignment Operators

- Default assignment operator
 - Allow class object **assignment**
 - Automatically overload **an assignment operator** for a class

```
Class_name & Class_name::operator=(const Class_name &);
```
- When an assignment operator is used and what it does
 - Assign one object to another **existing** object
 - **Not** used when **initializing** an object (copy constructor)
- What a default assignment operator does (the same to copy)
 - Perform a **member-to-member copy**
 - **Static** data members are **unaffected**



Default Assignment Operators

- Where assignment goes wrong
 - Cause both pointers to point to **the same address**
 - Attempt to **delete** the previously **deleted string**
 - How about **static** variables?
- Fixing assignment
 - Be **similar** to that of the **copy constructor** but have **differences**
 - ✓ **Proceed to free** the memory (existing object worked for previous tasks)
 - ✓ Protect against **assigning one to itself**
 - ✓ Return a **reference** to the invoking object

```
S0 = S1 = S2;
```

```
S0.operator=(S1.operator=(S2));
```

- Revisit program

```
StringBad & StringBad::operator=(const StringBad & st)
{
    if (this == &st)           // object assigned to itself
        return *this;         // all done
    delete [] str;             // free old string
    len = st.len;
    str = new char [len + 1];   // get space for new string
    std::strcpy(str, st.str);   // copy the string
    return *this;              // return reference to invoking object
}
```

The New, Improved String Class



Add More Capabilities to the Class

- Add following methods

	<code>int length () const { return len; }</code>
Comparison	<code>{ friend bool operator<(const String &st, const String &st2); friend bool operator>(const String &st1, const String &st2); friend bool operator==(const String &st, const String &st2);</code>
Print	<code>friend operator>>(istream & is, String & st);</code>
Accessing	<code>{ char & operator[] (int i); const char & operator[] (int i) const;</code>
Static member	<code>static int HowMany();</code>

- Run `sayings1.cpp`, `string1.h`, `string1.cpp`



Comparison Members

- Use the standard `strcmp()` function
 - Return a **negative** value if its first argument **precedes** the second alphabetically
 - Return **0** if the strings are the **same**
 - Return a **positive** value if the first **follows** the second alphabetically
- And use built-in `<` operator

```
bool operator<(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) < 0);
}

bool operator>(const String &st1, const String &st2)
{
    return st2 < st1;
}

bool operator==(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) == 0);
}
```



Accessing Characters by Using Bracket Notation

- A standard C-style string

- Use brackets to access individual characters

```
char city[40] = "Amsterdam";
```

```
cout << city[0] << endl; // display the letter A
```

- In C++, the **two bracket symbols** constitute **a single operator**

- Place one operand in **front** of the first bracket

- Place the other operand **between** the two brackets

- Declaring the return type as type **char &** allows you to **assign** values to a particular element

```
char & String::operator[](int i)
{
    return str[i];
}
```

```
String means("might");
means[0] = 'r';
```



Static Class Member Functions

- Declare a member function as being static
 - The keyword **static** should appear in the function **declaration** but not in the function definition
 - **Doesn't** have to be invoked by an **object**
 - **Doesn't** get a **this** pointer to play with
 - **Public** static function can be invoked using the **class name** and the **scope-resolution operator**
 - The **only** data members it can use are the **static data members**

```
static int HowMany() { return num_strings; }
```

```
int count = String::HowMany(); // invoking a static member function
```



Further Assignment Operator Overloading

- Copy an **ordinary string** to a String object which is user-defined

```
String name;  
char temp[40];  
cin.getline(temp, 40);  
name = temp; // use constructor to convert type
```

- If there is no op. overloading

- Three steps:

- ✓ Use the **String(const char *)** constructor to construct a temporary String object containing a copy of the string stored in temp
- ✓ Use the **String & String::operator=(const String &)** function to copy information from the temporary object
- ✓ Call the **~String()** destructor to delete the **temporary** object

- **Overload** the assignment operator

- One step

```
String & String::operator=(const char * s)  
{  
    delete [] str;  
    len = std::strlen(s);  
    str = new char[len + 1];  
    std::strcpy(str, s);  
    return *this;  
}
```



Things to Remember When Using **new** in Constructors

- Use **new** to initialize a pointer member in a constructor and use **delete** in the destructor
- The uses of new and delete should be **compatible**
- For multiple constructors, all should use new the **same way**
- Define a **copy** constructor that initializes one object to another
 - Copy the data, **not just the address** of the data (why?)
 - Update any **static** class members (why?)
- Define an **assignment** operator that copies one object to another
 - Copy the data, **not just the address** of the data (why?)
 - **Check** for self-assignment, **free** memory, and return a **reference** (why?)



Observations About Returning Objects

- Returning a const reference to a const object
 - The usual reason for using a reference is **efficiency**
 - **Doesn't** invoke the **copy** constructor
 - **Exist** when the calling function is executing
- Returning a reference to a non-const object
 - Overloading the **assignment** operator (efficiency)
 - Overloading the **<<** operator for use with **cout** (necessity)
- Returning an object (overloaded arithmetic operators)
 - **Local object should not** be returned by reference
- Run **sayings1.cpp, string1.h, string1.cpp**

```
Vector force1(50,60);  
Vector force2(10,70);  
Vector max;  
max = Max(force1, force2);
```

```
String s1("Good stuff");  
String s2, s3;  
s3 = s2 = s1;
```

```
String s1("Good stuff");  
cout << s1 << "is coming!";
```

```
Vector force1(50,60);  
Vector force2(10,70);  
Vector net;  
net = force1 + force2;
```

Using Pointers to Objects



Use New and Delete on Two Levels

- Run sayings2.cpp (.h, .cpp files in the previous example)

- Member pointers (scope-resolution operator)

- ✓ Allocate storage **space** for **each object** that is created
- ✓ Happen in the **constructor** functions
- ✓ Destructor functions use **delete** to free that memory with brackets

- Pointers point to objects (dereference)

```
String * favorite = new String(sayings[choice]);
```

- ✓ Provide the only access to the **nameless** object created by **new**
- ✓ **Constructors** allocate space and assign the address to member **pointer**
- ✓ Use **delete** to delete this object when it is finished with it
- ✓ **Static** member is **stored separately** from the objects



Destructor Takes Care of the Final Task

- Destructors are called in the following **situations**
 - If an object is an automatic variable, the object's destructor is called when the **program exits the block**
 - If an object is a static variable (external, static, static external, or from a namespace), its destructor is called when the **program terminates**
 - If an object is created by **new**, its destructor is called only when you **explicitly use delete** on the object

```
class Act { ... };  
...  
Act nice; // external object  
...  
int main()  
{  
    Act *pt = new Act; // dynamic object  
    {  
        Act up; // automatic object  
        ...  
    }  
    delete pt;  
    ...  
}
```

destructor for automatic object up
called when execution reaches end
of defining block

destructor for dynamic object *pt
called when delete operator
applied to the pointer pt

destructor for static object nice
called when execution reaches end
of entire program



Summary 1 of Pointers for Objects

- Pointers and objects

Declaring a pointer to a class object:

```
String * glamour;
```

Initializing a pointer to an existing object:

```
String * first = String object&sayings[0];
```

Initializing a pointer using new and the default class constructor:

```
String * gleep = new String;
```

Initializing a pointer using new and the String(const char*) class constructor:

```
String * glop = new String("my my my");
```

Initializing a pointer using new and the String(const String &) class constructor:

```
String * favorite = new String objectString(sayings[choice]);
```

Using the -> operator to access a class method via a pointer:

```
if (objectsayings[i].length() < pointer to objectshortest->length())
```

Using the * dereferencing operator to obtain an object from a pointer:

```
if (objectsayings[i] < object*pointer to objectfirst)
```



Summary 2 of Pointers for Objects

- Creating an object with **new**

```
String *pveg = new String("Cabbage Heads Home");
```

1. Allocate memory for object: _____

str: _____
len: _____

Address: 2400

2. Call class constructor, which

- allocates space for "Cabbage Heads Home"
- copies "Cabbage Heads Home" to allocated space
- assigns address of "Cabbage Heads Home" string to string to str
- assigns value of 19 to len
- updates num_strings (not shown)

Cabbage Heads Home\0

Address: 2000

str: 2000
len: 19

Address: 2400

3. Create the pveg variable: _____

pveg – Address: 2800

4. Assign address of new object to the pveg variable: _____

2400

pveg – Address: 2800



Looking Again at Placement new

- Placement **new**
 - Allow you to **specify** the memory location used to allocate memory
- Run **placnew1.cpp**
 - Problems
 - ✓ Placement **new overwrites** the **same** location used for the first object with a new one
 - ✓ Using **delete []** with **buffer does not** invoke the **destructors** for the objects created with placement new
- Solution 1
 - Manage the memory **locations**

```
pc1 = new (buffer) JustTesting;  
pc3 = new (buffer + sizeof (JustTesting)) JustTesting("Better Idea", 6);
```



Arrange for Destructors for Placement new

- Reasons of the second problem:
 - `delete` works in conjunction with `new` but not with placement new
 - `pc3` does not receive an address returned by `new`
 - `pc1` has the `same` numeric value as `buffer`
 - `delete [] buffer` `doesn't` call the `destructors` for any objects
- Solution 2: arrange for the destructors to be called
 - Call the destructor `explicitly` for any object created by placement new
- Run `placnew2.cpp`

```
pc3->~JustTesting(); // destroy object pointed to by pc3  
pc1->~JustTesting(); // destroy object pointed to by pc1
```




Reviewing Techniques

- Overloading the `<<` operator

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ; // display object contents
    return os;
}
```

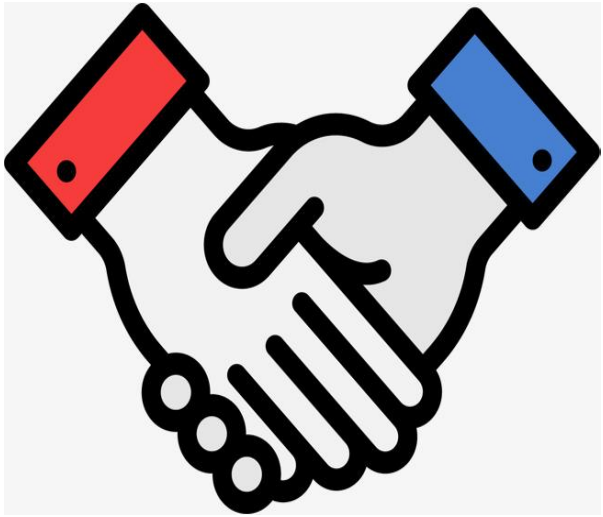
- Conversion functions

- Convert a **single** value to a class type `c_name(type_name value);`
- Convert a **class type** to **some other types** `operator type_name();`
- Keyword **explicit** when declaring a constructor to **prevent** it from being used for **implicit conversions**



Reviewing Techniques

- Classes whose constructors use **new**
 - Any class member that points to memory allocated by **new** should have the **delete** operator applied to it in the class destructor
 - If a destructor frees memory by applying delete to a pointer, every constructor should **initialize that pointer**, either by using **new** or by setting the pointer to the **null** pointer
 - Constructors should use either **new []** or **new**, but not a mixture
 - Define a copy constructor that **allocates new memory** rather than **copying a pointer** to existing memory
 - Define a class member function that **overloads** the **assignment** operator



Thanks



zhengf@sustech.edu.cn