Lec_21.md

Lecture 21 - An Abstract String Class

Oct. 28/2020

Previously, we've dealt with objects with **pointer** data members

- Let's implement a custom/abstract `String` class as an example
  - `String`, uppercase S
    - Different from built in `string`

String Class

Want to be able to do:

Create Strings

```
String FirstName("Tarek");
String LastName = "Abdelrahman";
String name;
```

Notes:

1. `String FirstName("Tarek")`

- `"Tarek"` is a c-string
  - C style, **null-terminated** string

Access Strings

```
FirstName.length();
char c = LastName[0];
FirstName[3] = 'i';
```

Notes:

1. `FirstName.length();`

- Get length of string

2. `FirstName[3] = 'i';`

- Modify and access strings as if they were arrays

Operate on Strings

```
if(FirstName=="Tarek"){ ... }
if(LastName==FirstName){ ... }
if(FirstName < LastName){ ... }
FirstName = name;
```

Notes:

1. `if(FirstName<LastName);`

- Compare strings

Print Strings

```
cout << FirstName;
```

String Object

Need to have:

Data Members

1. `char* str`

- Character array just large enough to hold characters of string
- **Null-terminated**
  - Contains a `'\0'` character to indicate end of string
- `char* str` is **shallow** data
  - The value pointed to by `char str*` should be **deep** data

2. `int len;`

- Contains length of string

Initial Class Definition

```
class String{
private:
  char* str;
  int len;
public:
  ...
};
```

Constructors

1. `String();`

- Default Constructor

2. `String(const char *s);`

- C-string Constructor

3. `String(const String & s);`

- Copy constructor

Constructors must dynamically allocate data to hold characters of string

- Initialize dynamically allocated data (**deep** data)

Default Constructor

```
String::String(){
  len=0;
  str = new char[1];
  str[0] = '\0';
}
```

C-String Constructor

```
String::String(const char* s){
  len = strlen(s);
  str = new char[len+1];
  strcpy(str,s);
}
```

Copy Constructor

```
String::String(const String & s){
  len = s.len;
  str = new char[len+1];
  strcpy(str,s,str);
}
```

Default Destructor

```
String::~String(){
  delete [] str;
}
```

Accessors

What if we want to find

- Length of the string
- Character at specific index

length()

```
int String::length() const{
  return len;
}
```

To get the character of a `char* array` at a specific index, we can call

- `arrayName[i]` for i in bounds
  - But what can we do for objects?
    - **Overload** the `operator[]` operator!

Operators

operator[]

```
char & String::operator[](int i){
  if((i<0)||(i>len-1)){
    cerr << "Error: out of bounds";
    exit(0);
  }
  return str[i];
}
```

Notes:

1. `char & String::operator[](int i){ }`

- Notice the *return by reference*
  - The character itself is returned by `operator[]`
  - Not just a copy of the character
- The usage of this operator is `ObjectName[i]`
  - The array index `i` is the parameter `i` in `operator[](int i)`

2. `if((i<0)||(i>len-1))`

- Check if `int i` param is in bounds

3. `cerr`

- Similar to `cout`, but used for errors
- Not a great implementation
  - Would rather *throw an exception* (and let that be handled)

operator=

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

Notes:

1. `if (this == &rhs) return (*this);`

- Imagine if we called `fname=fname`
  - `delete [] str;` woud *delete* the value that we are trying to copy a few lines later.
  - Need some guard in case we try to assign an object to itself.
- The `&` is an **overloaded symbol**
  - In the function definition `const String & rhs`, `&` defines the pass type for parameters
  - In `this == &rhs`, `&` is the reference operator
  - Standalone, `&` is the bitwise AND operator
  - Paired up, `&&` is the logical AND operator

2. `delete [] str;`

- Delete the original string stored by `this`

3. `len = rhs.len;`

- First copy the length of `rhs` to `this`

4. `return (*this);`

- Return the lhs of the `=` call, or the `this`

Comparison Operators

Define:

1. `operator<`
2. `operator>`
3. `operator==`

operator< and operator>

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

Notes:

1. `strcmp(str,rhs.str)<0`

- Using the built in C library functions
- `strcmp` compares objects **lexicographically**
  - Based on order in the alphabet

2. Pretty much the same definition for `operator<`

operator==

There are multiple cases for the definition of `operator==` . Compare:

1. `fname == "Stewart";`
2. `"Tarek" == FirstName;`
3. `fname == FirstName;`

So need to define multiple `operator==` calls

Comparing String and String

Comparing two objects of class `String` :

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

Comparing String and const char*

```
bool String::operator==(const char* s) const {
  return (strcmp(str, s) == 0);
}
```