HW2 report

2012-11114
경영학과
엄태휘

ostream& operator<<(ostream &out, const String &str)
out 에 String class의 sPtr을 넘겨준 뒤 에, out을 반환하였다.
Friend member function이기 때문에 String::operator<<로 할 필요가 없다.

```
int String::calLength(const char *str)
{
  int cnt=0;

  while (true) {
    if (str[cnt]!='\0') {
      cnt++;
    }
    else
      break;
  }

  return cnt;
}
```

String 클래스의 sPtr에 동적할당을 하기 위해서, 입력받은 문자열의 길이를(null문자는 제외) 계산해주는 함수이다. null문자를 만날 때까지 카운터의 값을 1씩 더해주어 문자열의 길이를 계산한다.

```
String::String(const char *s)
{
  length = calLength(s);
  //add 1 for null character
  sPtr = new char[length+1];
  for (int i = 0; i < length+1; i++) {
    this->sPtr[i] = s[i];
  }
}
```

calLength 를 통해 문자열의 길이를 계산 한 후에, sPtr에 메모리를 동적으로 할당해준다음, 문자열s의 각 글자를 sPtr에 복사한다.
문자열 대신, 스트링 클래스를 받는 copy constructor의 경우에도 비슷한 방법으로 스트링 클래스의 length를 s.length를 통하여 받은 후에 한다.

```
String::~String()
{
  delete[] sPtr;
  sPtr = NULL;
}
```

new로 할당받은 sPtr을 delete[]를 통하여 해제한 후에, sPtr가 NULL을 가르키도록 한다.

```
const String& String::operator=(const String& str)
{
  //same as copying
  this->length = str.length;
  char buf[length+1];

  for (int i = 0; i < length+1; i++) {
    buf[i] = str.sPtr[i];
  }
  delete[] sPtr;
  sPtr = new char[length+1];

  for (int i = 0; i < length+1; i++) {
    sPtr[i] = buf[i];
  }

  return *this;
}
```

= 연산자는 copy연산자와 거의 동일하다.
다만 self assignment를 할 때 sPtr = new char[length+1]를 하게 되면 원래의 값이 사라지므로 buf에 잠시 담아둔다.

```
const String& String::operator+=(const String &str)
{
  char buffer[length];
  //we don't need a null character just a character buffer, not a string.
  int tmpLen = length;
  this->length += str.length;
  int i;

  //copy string to temporary buffer.
  for (i = 0; i < tmpLen; i++) {
    buffer[i] = sPtr[i];
  }

  //re-allocate sPtr with bigger size.
  delete[] sPtr;
  sPtr = new char[length+1];

  //copy from temp buffer to original.
  for (i = 0; i < tmpLen; i++) {
    sPtr[i] = buffer[i];
  }

  //add str to original string
  for (i = tmpLen; i < length; i++) {
    sPtr[i] = str.sPtr[i-tmpLen];
  }
  sPtr[i] = '\0';

  return *this;
}
```

원래 스트링의 길이만큼 임시버퍼를 생성한 다음(문자열로 다룰 것이 아니라, 글자만 복사하면 되므로 null character만큼 +1을 해주지 않아도 된다.) 원래의 문자들을 버퍼에 복사한다. 그 후에, 스트링을 원래의 길이+ 이어 붙일 스트링의 길이만큼 할당 받은 다음 버퍼의 문자들과, str의 문자열을 복사한다.
또한, 메모리의 누수를 막기 위해 sPtr을 새로이 할당하기 전에 delete[] sPtr을 해준다.

! 연산자:
문자열을 생성할 때마다 길이를 바꿔주므로, 길이가 0이라면 비어있는 스트링, 길이가 1이상이라면 비어있지 않은 스트링임을 알 수 있다.

```
bool String::operator==(const String &s) const
{
  bool isEqual = true;
  int i;

  if (this->length != s.length) {
    //if different length, we don't have to look at the contents
    isEqual = false;
  }
  else {
    for (i = 0; i < this->length; i++) {
      //if there is different character
      if(this->sPtr[i] != s.sPtr[i]) {
        isEqual = false;
        break;
      }// endif
    }
  }// endif

  return isEqual;
}
```

==연산자의 경우, isEqual의 기본값을 일단 true로 한다. 그 후에, 두 문자열의 길이가 다르다면 내용을 확인할 필요도 없이 다르므로 isEqual을 false로 설정한다. 그리고 두 문자의 길이가 같다면, 두 문자열의 같은 위치에서 다른 글자가 나올때까지 for문을 돈다. 다른 글자가 나온다면, isEqual = false로 하고 포문에서 나온다. 만약 모든 글자가 다 같다면 isEqual의 값이 바뀌지 않으므로 true가 반환된다.

```
bool String::operator<(const String &s) const
{
  bool comesFirst = false;
  int i;

  for (i = 0; i < myMin(this->length,s.length); i++) {
    if(this->sPtr[i] < s.sPtr[i]) {
      comesFirst = true;
      break;
    }// endif
    else if(this->sPtr[i] > s.sPtr[i]) {
      comesFirst = false;
      break;
    }// endif
  }
  //if same till the end, default value(false) will be returned.
  return comesFirst;
}
```

< 연산자의 경우, 기본값을 일단 false로 한다. 그 후에 각 문자열의 같은 위치에 있는 글자들을 비교한다. 만약 왼쪽 문자열의 글자의 사전상 순서가 오른쪽 문자열의 글자의 사전상 순서보다 빠르다면, comesFirst 가 true가 되고, 그 반대의 경우에는 comesFirst 가 false가 된다.

만약 모든 글자가 같다면, 기본값인 false가 출력된다.

```
bool String::operator>(const String &s) const
{
  bool comesLater=false;
  int i;

  for (i = 0; i < myMin(this->length,s.length); i++) {
    if(this->sPtr[i] < s.sPtr[i]) {
      comesLater = false;
      break;
    }// endif
    else if(this->sPtr[i] > s.sPtr[i]) {
      comesLater = true;
      break;
    }// endif
  }
  //if same till the end, default value(false) will be returned.
  return comesLater;
  //vs
  //easier but slower
  //return (!(*this < right) && !(this == right));
}
```

연산자 >는 연산자 <와 구조는 동일하지만, 다만 글자의 사전상 순서에 따른 boolean값이 반대가 될 뿐이다. Return (!(*this < s) && !(*this== s)) 를 하면 더 간단하지만, <를 구할 시간에 ==인지도 확인해야 하므로 조금 더 비효율적이 된다.

```
// < or ==
bool String::operator<=(const String &right) const
{
  return (*this < right || *this == right);
}

// > or ==
bool String::operator>=(const String &right) const
{
  return (*this > right || *this == right);
}
```

<= 이나 >=은 (< or ==), (> or ==) 의 결과값을 반환하도록 하면 된다.

```
char& String::operator[](int a)
{
  try {
    if(a < 0 || a > this->length-1)
      throw a;
  }
  catch(int exception) {
    std::cout << "Array Exception: index " <<
      exception << " is out of range." << std::endl;
  }

  return (this->sPtr[a]);
}
```

[] 연산자의 경우, int a가 범위 밖에 있는 지를 try catch 문으로 확인한 뒤에, 범위 밖에 있지 않다면 sPtr[a]를 반환하도록 한다.
const의 경우, 주소값이 아닌 문자의 값만을 반환하도록 한다.

()연산자의 경우, 우선 try catch로 요구한 substring의 시작(int start)과 끝(int end)이 범위 안에 있는지 확인한다.
그 후,

```
s.length = (start<=end? end-start+1 : length-start +1);
```

로 end 가 0이 아니라면 end - start +1만큼 할당, end가 0이라면 start부터 끝까지 출력할 수 있도록 메모리를 할당한다.
그 후 s의 sPtr에 문자를 복사한다.

*기타
범위 밖의 값을 바꾸려고 할 경우, []에서 에러문을 출력하긴 하지만, 프로그램에서 = 연산자를 통해 여전히 그 값을 할당하게 된다. 그런데 리눅스에서 s1[30]같은 작은 값을 입력할 경우, 메모리주소에서 인접한 s2 s3등의 값을 바꾸게 되어 free하는 과정에서 s1을 free할 때 다른 sPtr도 free하게 되어 double free에러가 발생한다. 그래서 s1[300]의 큰 값으로 바꾸어 빈 공간의 값을 바꾸도록 하여 double free에러가 발생하지 않게끔 하였다.