Chapter

16

프로젝트 1 : STL 활용하기

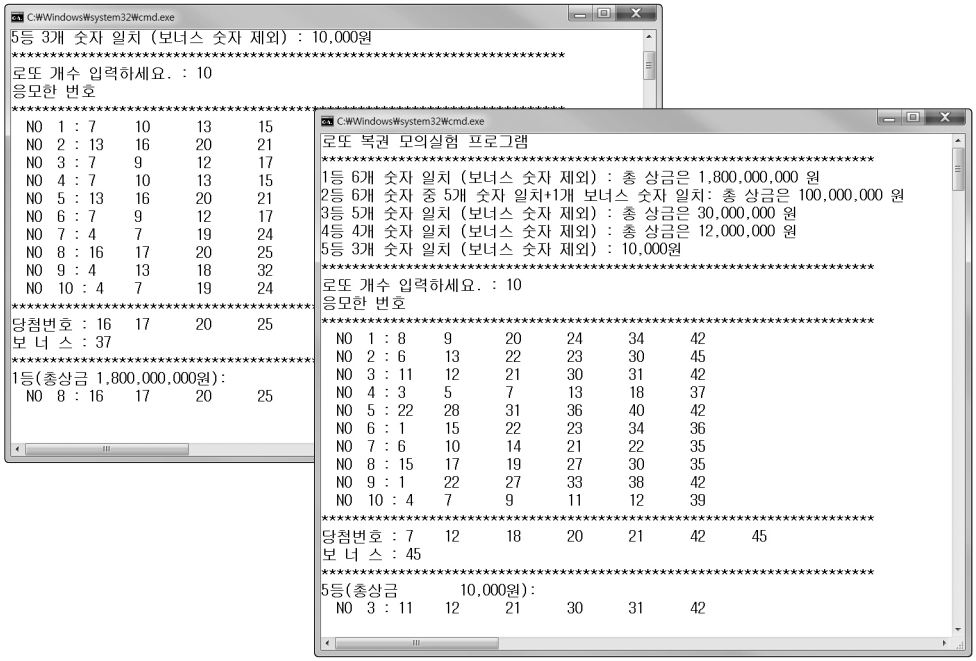
✻ **학습목표**

- 로또 복권 모의실험 프로그램을 작성한다.
- 학생 성적 관리 프로그램을 작성한다.

Section

01 로또 복권 모의실험 프로그램

STL을 학습하면서 배열 대용으로 벡터를 사용하는 것을 알았다. 이번 프로젝트를 통해 STL의 활용 방법을 살펴보려고 벡터를 사용해서 로또 복권 모의실험 프로그램을 작성하려고 한다. 먼저 구입할 로또 개수를 입력 받는다(여기서는 10개라고 가정했다). 그리고 당첨 번호 6개와 보너스 번호 하나를 발표한다. 마지막으로 로또 당첨 여부와 사용자가 실제로 받게 될 상금을 출력한다.



로또는 1~45 사이의 번호를 중복되지 않게 무작위로 숫자 6개를 뽑아서 구한다. 응모한 번호와 함께 등수를 저장하기 위해서 클래스를 선언하고, 당첨 번호는 벡터에 저장한다.

예제 16-1 로또 복권 모의실험(16_01.cpp)

```
001 #include <vector>
002 #include <set>
003 #include <algorithm>
004 #include <time.h>
005 #include <iostream>
006 using namespace std;
007
008 void PrintElement(int  n)
009 {
010   cout<<n<<"\t";
011 }
012
013 class LottoGenerator {
014 public:
015   vector<int> ball;
016
017   LottoGenerator(int count)
018   {
019     makeRandom(count);
020   }
021
022   LottoGenerator()
023   {
024     makeRandom(6);
025   }
026
027   set<int> RangedRandDemo( int range_min, int range_max, int n)
028   {
029     set <int> s;
030     while(true)
031     {
032       int u = (double)rand() / (RAND_MAX + 1) *
033             (range_max - range_min)  + range_min;
034       s.insert(u);
035       if(s.size() >= n)
036          break;
037     }
038     return s;
```

```
039   }
040
041   void makeRandom(int count){
042     set<int> s;
043     s=RangedRandDemo( 1, 46, count);
044
045     set<int>::iterator iter;
046     for(iter=s.begin(); iter!=s.end(); iter++)
047       this->ball.push_back(*iter);
048   }
049
050   void lottprn() {
051     for_each(this->ball.begin(),this->ball.end(), PrintElement) ;
052     cout <<endl;
053   }
054 };
055
056 class UserLotto {
057 public:
058   int pos;
059   LottoGenerator lotto;
060   int grade;
061
062   UserLotto()
063   {
064     grade=0;
065   }
066 };
067
068 void line()
069 {
070   cout<<"**************************************************************************\n";
071 }
072
073 void title()
074 {
075   cout<<"로또 복권 모의실험 프로그램\n";
076   line();
077   cout<<"1등 6개 숫자 일치 (보너스숫자 제외):총 상금은 1,800,000,000원\n";
```

```
078   cout<<"2등 6개 숫자 중 5개 숫자 일치+1개 보너스 숫자 일치:";
079   cout<<"총 상금은 100,000,000원\n";
080   cout<<"3등 5개 숫자 일치 (보너스 숫자 제외):총 상금은 30,000,000원\n";
081   cout<<"4등 4개 숫자 일치 (보너스 숫자 제외):총 상금은 12,000,000원\n";
082   cout<<"5등 3개 숫자 일치 (보너스 숫자 제외):10,000원\n";
083   line();
084 }
085
086 void Lotto(UserLotto user)
087 }
088   switch(user.grade){
089     case 1:
090       cout<<"1등(총상금 1,800,000,000원):";
091       break;
092     case 2 :
093       cout<<"2등(총상금   100,000,000원):";
094       break;
095     case 3 :
096       cout<<"3등(총상금    30,000,000원):";
097       break;
098     case 4 :
099       cout<<"4등(총상금    12,000,000원):";
100       break;
101     case 5 :
102       cout<<"5등(총상금       10,000원):";
103       break;
104   }
105   if(user.grade>=1  && user.grade<=5){
106     cout<<"\n  NO  "<<user.pos<<" : ";
107     user.lotto.lottprn();
108   }
109 }
110
111 void counts(vector<UserLotto> user, LottoGenerator lotto)
112 {
113   int matchingcount;
114   vector<int>::iterator num1, num2;
115   for(int i=0; i<user.size(); i++){
116     matchingcount=0;
```

```
117     for(num1=lotto.ball.begin(); num1<lotto.ball.end()-1; num1++){
118       for(num2=user[i].lotto.ball.begin();num2<user[i].lotto.ball.end();num2++){
119         if(*num1 < *num2){
120           break;
121         }
122         if(*num1 == *num2){
123           matchingcount++;
124         }
125       }
126     }
127
128     switch(matchingcount){
129       case 3:
130          user[i].grade=5;
131          break;
132       case 4:
133          user[i].grade=4;
134          break;
135       case 5:
136          {
137          user[i].grade=3;
138          int bonus=*(lotto.ball.end()-1);
139          vector<int>::iterator iter;
140          iter = find(user[i].lotto.ball.begin(),user[i].lotto.ball.end(),bonus);
141          if( iter != user[i].lotto.ball.end())
142            user[i].grade=2;
143          }
144          break;
145       case 6:
146          user[i].grade=1;
147          break;
148     default :
149          user[i].grade=0;
150     }
151     Lotto(user[i]);
152   }
153 }
154
155 void main()
```

```
156 {
157   vector<UserLotto> user;
158   int userCount;
159   int pos=1;
160   int i;
161
162   title();
163   cout<<"로또 개수 입력하세요. : ";
164   cin>>userCount;
165
166   cout<<"응모한 번호 \n";
167   line();
168   for(i=0; i<userCount; i++) {
169     UserLotto uselotto;
170     uselotto.pos=i+1;
171     cout<<"  NO  "<<uselotto.pos<<" : ";
172     uselotto.lotto.lottprn();
173     user.push_back(uselotto);
174   }
175
176   LottoGenerator lotto(7);
177   line();
178   cout<<"당첨번호 : ";
179   lotto.lottprn();
180   cout<<"보 너 스 : "<<lotto.ball[6]<<endl;
181   line();
182
183   counts(user, lotto);
184 }
```

13행 로또 번호를 저장하는 클래스를 선언한다.

15행 로또 번호를 저장할 벡터를 선언한다.

27행 랜덤 함수로 1~45번 사이 번호 숫자를 무작위로 받아서 서로 다른 수 6개가 나올 때까지 임의의 수를 만든다.

50행 번호 6개를 출력하는 함수다.

56행 응모한 번호를 저장하는 클래스를 선언한다.

59행 구입한 로또 객체를 선언한다.

60행 등수를 저장할 변수를 선언한다.

82행 사용자가 실제적으로 받게 될 상금을 출력한다.

111행 응모한 번호와 당첨 번호가 일치되는 개수로 등수를 구하는 함수다.

113행 응모한 번호와 당첨 번호가 일치되는 개수를 저장할 변수를 선언한다.

117행 일치되는 개수를 구하려고 당첨 번호 중 6개만 가지고 반복문을 돌린다.

118행 일치되는 개수를 구하려고 응모한 번호 6개의 수치로 반복문을 돌린다.

135행 5개가 일치되었다면 보너스 숫자를 비교하여 2등, 3등을 결정한다.

157행 구입한 로또를 저장할 벡터를 선언한다.

158행 구입한 로또의 개수를 저장할 변수를 선언한다.

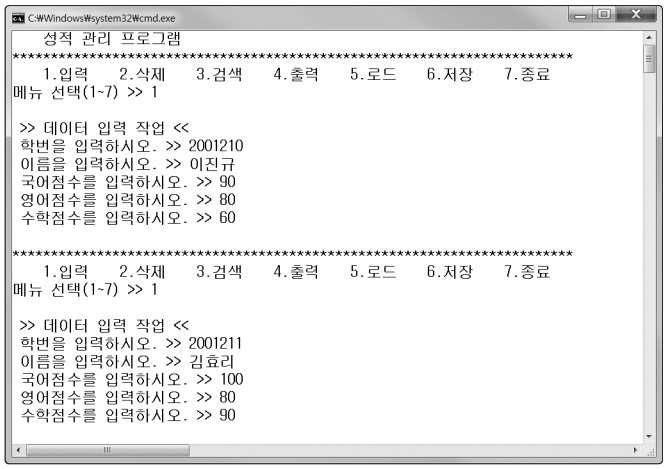
176행 '당첨 번호+보너스 번호'를 서로 다른 랜덤값으로 구해서 저장하는 로또 객체다.

Section

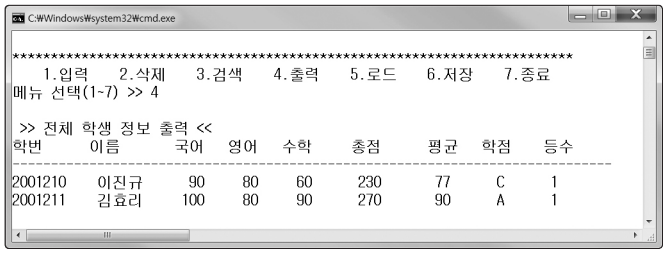
02 학생 성적 관리 프로그램

학생 성적 관리에 필요한 데이터를 키보드에서 입력받아 성적을 처리한 후 이를 파일에 저장해 두는 프로그램이다. 학생 성적 관리 프로그램은 입력, 삭제, 검색, 출력, 로드, 저장, 종료의 7가지 작업 중 하나를 선택하도록 메뉴로 작성되어 있다. 학생 성적 관리 프로그램은 벡터로 작성할 것이다. 우선 성적 관리를 위한 클래스 Student를 사용한다.

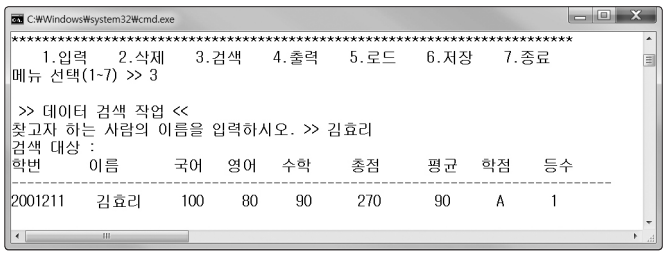
'1.입력' 메뉴를 선택하고 성적관리에 필요한 데이터를 키보드에서 입력받는다.



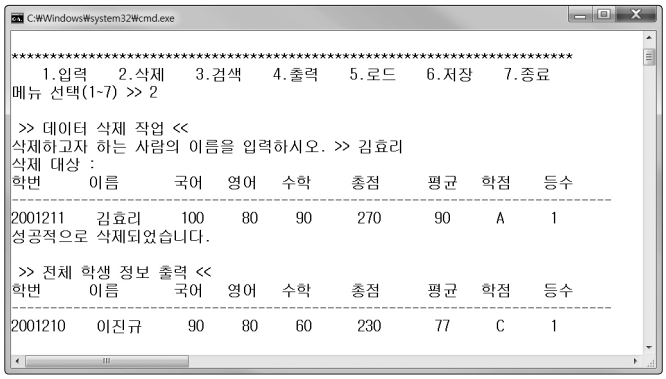
'4.출력' 메뉴는 전체 학생 정보를 출력한다.



'3.검색' 메뉴는 이름 정보를 검색한다.



'2.삭제' 메뉴는 이름을 입력받아 해당하는 정보를 삭제한다.

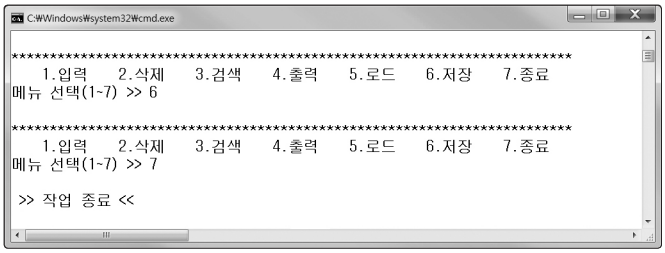


레코드 단위로 파일 입출력을 하려면 read(), write() 함수를 사용해야 한다. 벡터에 연결되어 있는 정보를 파일에 받게 하려면 유의할 점이 있다. 연결 리스트는 배열처럼 연속된 기억공간을 할당받지 않기 때문에 처음 노드부터 끝 노드까지 순회하면서 레코드 한 개 단위로 파일 출력을 해야 한다.

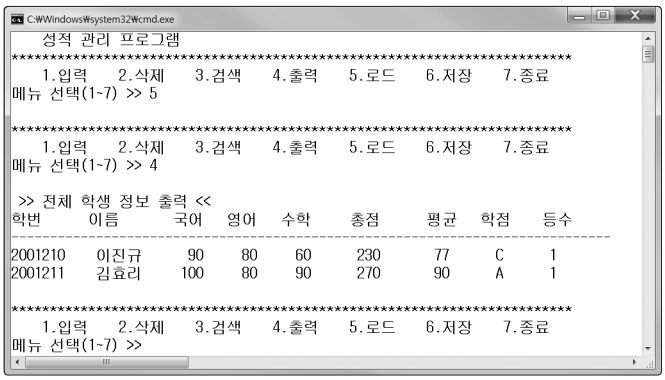
파일에 저장된 데이터를 읽어오는 함수를 구현할 때에는 현재 메모리에 있는 벡터의 요소를 모두 삭제해줘야 한다. 삭제하지 않고 디스크에서 읽어오게 되면 이전 벡터에 추가로 데이터가 연결되기 때문이다.

모든 요소를 삭제하는 함수인 delete_all 함수를 정의해서 파일에서 레코드를 읽기 시작하기 전에 미리 호출해준다. 그리고 read() 함수로 레코드 한 개 단위로 데이터를 읽어서 벡터에 연결한다.

우선 2개의 레코드를 입력한 후 '6.저장' 메뉴를 선택하여 파일에 저장하고 '7.종료' 메뉴를 눌러서 프로그램을 종료한다.



다시 프로그램을 실행한 후 '5.로드' 메뉴를 선택하여 파일에 저장된 데이터를 읽어온 후에 '4.출력' 메뉴를 눌러서 이전에 프로그램을 실행했을 때 입력해 두었던 레코드가 다시 출력되는 것을 확인한다.



예제 16-2 학생 성적 관리 프로그램(16_02.cpp)

```
001 #include<vector>
002 #include <algorithm>
003 #include <iostream>
004 #include <iomanip>
005 #include <fstream>
006 using namespace std;
007
008 // 성적 저장을 위한 클래스 정의
009 class Student {
010 private:
```

```
011   int no;              // 학번
012   char name[256];      // 이름
013   int kor, eng, math;  // 국어, 영어, 수학 점수
014   int total;           // 총점
015   double avg;          // 평균
016   char level;          // 학점
017   int  grade;          // 등수
018
019 public :
020   Student(int no, char * name, int kor, int math, int eng);
021   Student(char * name);
022   Student();
023   void calculate() ;
024   friend ostream &operator<<(ostream &os, const Student &stu);
025   friend void PrintElement(Student stu) ;
026   bool operator==(const char * key) const;
027   void getStudent();
028 };
029
030 Student::Student(int no,  char * name, int kor, int math, int eng)
031 {
032   this->no=no;
033   strcpy(this->name, name);
034   this->kor=kor;
035   this->math=math;
036   this->eng=eng;
037   calculate();
038 }
039
040 Student::Student(char * name)
041 {
042   strcpy(this->name, name);
043 }
044
045 Student::Student()
046 {
047   calculate();
048 }
049
```

```
050 void Student::calculate()
051 {
052   total = kor + eng + math;
053   avg = total / 3.0;
054   switch(int(avg)/10){
055     case 10:
056     case 9:  level ='A';  break;
057     case 8:  level ='B';  break;
058     case 7:  level ='C';  break;
059     case 6:  level ='D';  break;
060     default : level ='F';
061   }
062   grade=1;
063 }
064
065 ostream &operator<<(ostream &os, const Student &stu)
066 {
067   os<<setw(4)<<stu.no<<setw(10)<<stu.name<<setw(8)<<stu.kor
068     <<setw(7)<<stu.eng<<setw(7)<<stu.math<<setw(9)<<stu.total
069     <<setw(9)<<setprecision(2)<<stu.avg<<setw(7)<<stu.level
070     <<setw(7)<<stu.grade;
071   return os;
072 }
073
074 bool Student::operator==(const char * key) const
075 {
076   return strcmp(name, key)==0;
077 }
078
079 void Student::getStudent()
080 {
081   cout<<"\n 학번을 입력하시오. >> ";
082   cin>>no;
083   cout<<" 이름을 입력하시오. >> ";
084   cin>>name;
085   cout<<" 국어점수를 입력하시오. >> ";
086   cin>>kor;
087   cout<<" 영어점수를 입력하시오. >> ";
088   cin>>eng;
```

```
089   cout<<" 수학점수를 입력하시오. >> ";
090   cin>>math;
091   fflush(stdin);
092   calculate();  // 총점, 평균, 학점을 구함
093 }
094
095 void PrintElement(Student stu)
096 {
097   cout<<stu<<endl;
098 }
099
100 void title()
101 {
102   printf("학번  이름  국어  영어  수학  총점  평균  학점  등수 \n");
103   printf("------------------------------------------------------------------- \n");
104 }
105
106 char *smenu[] = {  "1.입력", "2.삭제", "3.검색",
107                    "4.출력", "5.로드", "6.저장", "7.종료"};
108
109 char select_menu(void)
110 {
111   char ch;
112   int i;
113   cout<<"\n*********************************************************\n";
114   for(i=0; i<sizeof(smenu)/sizeof(smenu[0]); i++){
115     cout<<setw(10)<<smenu[i];  //메뉴 출력
116   }
117   do{
118     printf("\n메뉴 선택(1~7) >> ");
119     cin.get(ch);
120     fflush(stdin);
121   }while(ch<'1' || ch>'7');
122   return ch;
123 }
124
125 class StudentManager{
126 public:
127   vector<Student> v;
```

```
128
129  void Input()  // 입력 함수 만들기
130  {
131    cout<<"\n >> 데이터 입력 작업 <<";
132    Student temp;
133    temp.getStudent();
134    v.push_back(temp);
135  }
136
137  void Display()
138  {
139    cout<<"\n >> 전체 학생 정보 출력 <<\n";
140    title();
141    for_each(v.begin(), v.end(), PrintElement) ;
142  }
143
144  void Delete()
145  {
146    char searchName[256];
147
148    cout<<"\n >> 데이터 삭제 작업 <<";
149    cout<<"\n삭제하고자 하는 사람의 이름을 입력하시오. >> ";
150    cin>>searchName;
151    fflush(stdin);
152
153    vector<Student>::iterator EraseIter = find(v.begin(), v.end(), searchName);
154
155    if (EraseIter != v.end()) {
156      cout<<"삭제 대상 :"<<endl;
157      title();
158      cout<<*EraseIter<<endl;
159      v.erase(EraseIter);
160      cout<<"성공적으로 삭제되었습니다. "<<endl;
161      this->Display();
162    }else{
163      cout<<searchName<<"님이 존재하지 않습니다. "<<endl;
164    }
165  }
166
```

```
167  void Search()
168  {
169    char searchName[256];
170
171    cout<<"\n >> 데이터 검색 작업 <<";
172    cout<<"\n찾고자 하는 사람의 이름을 입력하시오. >> ";
173
174    cin>>searchName;
175    fflush(stdin);
176
177    vector<Student>::iterator FindIter = find(v.begin(), v.end(), searchName);
178
179    if (FindIter != v.end()) {
180      cout<<"검색 대상 :" <<endl;
181      title();
182      cout<<*FindIter<<endl;
183    }else{
184      cout<<searchName<<"님이 존재하지 않습니다. "<<endl;
185    }
186  }
187
188  void delete_all()
189  {
190    for(int i=v.size()-1; i>=0; i--){
191       v.erase(v.begin()+i);
192    }
193  }
194
195  void Exit()
196  {
197    printf("\n >> 작업 종료<<\n");
198    delete_all();
199  }
200
201  void Load()
202  {
203    delete_all();
204
205    ifstream fin;
```

```
206     fin.open("r.txt");
207
208     Student temp;
209
210     while(fin.read((char *)&temp, sizeof(temp))) {
211       v.push_back(temp);
212     }
213      fin.close();
214   }
215
216   void Save()
217   {
218     ofstream fout;
219     fout.open("r.txt");
220
221     vector<Student>::iterator iter;
222
223     for(iter=v.begin(); iter!=v.end(); iter++)    {
224       Student temp=*iter;
225       fout.write((char *)&temp, sizeof(temp));
226     }
227     fout.close();
228   }
229 };
230
231 void main()
232 {
233   StudentManager sm;
234   sm.Load();
235
236   char ch;
237   cout<<"    성적 관리 프로그램";
238   while(1) {
239     ch=select_menu();
240     switch(ch){
241       case '1':  sm.Input();   break;
242       case '2':  sm.Delete();  break;
243       case '3':  sm.Search();  break;
244       case '4':  sm.Display(); break;
```

```
245      case '5':  sm.Load();   break;
246      case '6':  sm.Save();   break;
247      case '7':  sm.Exit();   exit(0);
248    }
249   }
250  }
```

09행 성적 저장을 위한 클래스를 정의한다.

65행 객체 단위로 출력하기 위해서 《 연산자 함수를 정의한다.

74행 학생 객체가 동일한지를 == 연산자로 비교하기 위해서 == 연산자 함수를 정의한다.

79행 성적 정보를 키보드에서 읽어오는 함수를 정의한다.

109행 메뉴를 출력하는 함수를 정의한다.

125행 성적 객체를 이용하여 성적 관리를 하는 클래스를 정의한다.

Chapter

17

프로젝트 2 : Mystring 클래스 설계하기

* 학습목표

• C++에서 제공해 준 클래스인 string에 대해서 학습한다.
• 클래스 설계 방법을 학습하려고 클래스를 분석한다.
• string 클래스를 모델삼아 직접 MyString 클래스를 구현한다.

Section

01 C++에서 제공해 주는 string 클래스

이 절에서는 문자 배열의 단점을 극복하는 방법으로 C++에서 제공해주는 string 클래스를 설계해 본다. 그런데 string 클래스를 무턱대고 설계하기 전에 이미 C++에서 제공해주는 string 클래스를 먼저 살펴본 후, 이 클래스와 동일한 작업이 가능하도록 설계해봄으로써 클래스를 설계하는 방법의 전반에 대해서 학습할 것이다.

C++에서 문자열 상수는 큰따옴표로 둘러싸인 문자의 집합으로 널 문자로 끝난다. C++에서는 문자열을 저장하기 위한 기본 자료형을 제공하지 않으므로 char형 1차원 배열에 저장해야 한다.

```
char strA[100] = "Apple";
```

배열에 문자열을 저장하고 이를 다루는 작업은 매우 까다롭다. 위 예는 문자열 배열에 초깃값으로 문자열 상수를 지정했지만 문자열 배열을 선언한 후에는 대입 연산자로 문자열 상수를 저장할 수 없다.

```
char strB[100];
strcpy(strB, "Banana");
```

이렇게 문자 배열을 사용해서 문자열을 저장할 때는 여러 가지 제약이 있다. 이러한 제약을 해결하려고 문자열 관련 함수가 제공되는데, 위의 예처럼 이미 선언된 문자 배열에 새로운 문자열 상수를 저장하기 위한 함수가 strcpy다. 그리고 이런 문자열 관련 함수를 사용하려면 string.h 헤더파일을 포함시켜야 한다.

```
#include<string>
```

두 문자열을 연결할 때는 strcat 함수도 사용할 수 있다.

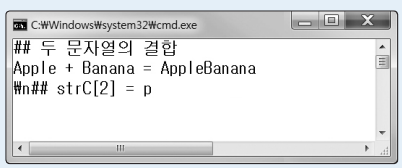
```
strcat(strA, strB);
```

위의 예와 같이 사용하면 strA에 저장된 “Apple” 다음에 strB에 저장된 “Banana”가 연결된다. 그래서 결국 strA에는 “AppleBanana”가 저장된다.

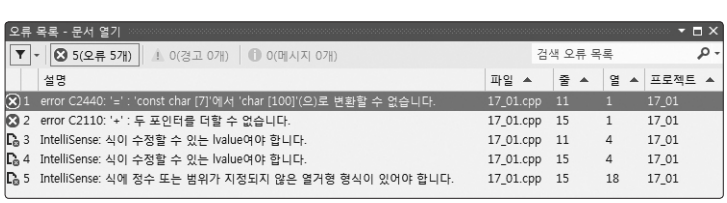
[예제 17-1]은 문자열 관련 함수를 사용해서 문자열을 문자 배열로 저장하는 프로그램이다.

예제 17-1 문자 배열에 문자열 저장하기(17_01.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 void main()
06 {
07   char strA[100] = "Apple";
08   char strB[100];
09   char strC[100];
10
11   // strB = "Banana";  // 에러
12
13   strcpy(strB, "Banana");
14
15   // strC = strA + strB;  // 에러
16
17   strcpy(strC, strA);
18   strcat(strC, strB);
19
20   cout<<"## 두 문자열의 결합\n";
21   cout<<strA<< " + " <<strB<< " = " <<strC<<"\n\\n";
22
23   cout<<"## strC[2] = "<<strC[2]<<"\n";
24 }
```



11행, 15행 주석 처리를 없애면 다음과 같은 에러가 발생한다.



13행 문자 배열에 문자열 상수를 대입할 때는 대입 연산자를 사용할 수 없고 strcpy 함수를 사용해야 한다.

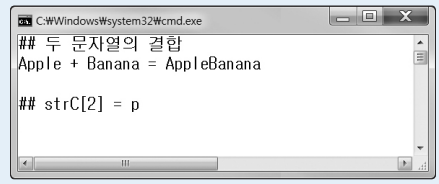
15행 + 연산자를 사용해서 두 문자열을 연결할 수 있는 프로그래밍 언어(예를 들어, 비주얼베이직)도 있지만 C++에서는 에러가 발생한다. 그러므로 strcat 함수를 사용해서 두 문자열을 결합해야 한다.

문자열을 문자 배열로 다룰 경우 매우 까다로운 문제들이 많기 때문에 C++에서는 string 클래스를 제공해서 문자열을 보다 간편하게 다룰 수 있도록 하고 있다. 예제를 통해 C++에서 제공하는 string 클래스의 사용법을 익혀보자.

[예제 17-2]는 string 클래스를 사용해서 문자열을 저장하는 프로그램이다.

예제 17-2 string 클래스를 사용해서 문자열 저장하기(17_02.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 void main()
06 {
07   string strA("Apple");
08   string strB;
09   string strC;
10
11   strB = "Banana";
12
13   strC = strA + strB;
14
15   cout<<"## 두 문자열의 결합\n";
16   cout<<strA<< " + " <<strB<< " = " <<strC<<"\n\n";
17
18   cout<<"## strC[2] = "<<strC[2]<<"\n";
19 }
```



07행 C++에서 string 클래스를 제공한다. 클래스로 객체 생성할 때 초깃값으로 문자열 상수 "Apple"을 지정하면 string 객체 strA에 "Apple"이 저장된다.

11행 08행에서 생성한 객체 strB에 대입 연산자로 "Banana"를 저장한다.

13행 string 객체 strA와 strB를 + 연산하면 두 string 객체에 저장된 문자열이 결합된다. 결합된 문자열을 strC에 저장한다.

18행 string 객체 strC를 배열처럼 [ ] 연산자를 사용해서 한 문자만 추출한다.

Section

2 MyString 클래스 기본 설계하기

이제 본격적으로 C++에서 제공해주는 string 클래스를 설계해 보자. 클래스를 새로 설계하는 이유는 string 클래스와 같은 기능을 하면서 문자 배열의 단점을 극복하기 위함이다. 그래서 새로 구현한 string 클래스는 문자열 배열에 없는 기능도 추가하고 문자열 배열에서 사용하던 기능까지도 재정의해 줄 것이다. 그럼, 먼저 극복해야 할 문자열의 단점과 해결 방법부터 살펴보자.

문자 배열의 단점

문자 배열은 정적 메모리 할당을 하기 때문에 문자열의 길이에 상관없이 고정된 메모리를 사용하므로 메모리가 낭비된다. 반대로 할당된 메모리를 지나서 문자열을 저장하게 될 수도 있으므로 위험하다.

MyString 클래스로 극복

우리가 설계할 MyString 클래스는 필요한 만큼의 메모리만 스스로 알아서 할당할 수 있도록 설계한다. 클래스가 생성될 때 초깃값에 의해 알맞은 메모리가 할당되도록 생성자를 구현한다. 생성되는 클래스마다 서로 다른 크기의 메모리를 할당받도록 하려면 동적 메모리 할당을 해야 한다.

MyString 클래스에 필요한 멤버변수

동적 메모리 할당을 하는 MyString 클래스를 구현하려면 문자열의 길이를 저장할 정수형의 멤버변수(int m_nLen)와 동적 메모리 할당한 힙 영역을 가리킬 문자 포인터형의 멤버변수(char *m_pStr)가 필요하다.

```
int m_nLen;
char *m_pStr;
```

MyString 클래스에 필요한 생성자 정의하기

어떠한 초깃값으로도 MyString 객체를 생성할 수 있도록 하려면 다음과 같은 2가지 종류의 생성자를 구현해야 한다.

[표 17-1] MyString 클래스의 생성자

생성자	설명
MyString(const char * const str)	문자열 상수나 문자열 배열을 초깃값으로 하는 생성자
MyString( )	기본 생성자로 널 값을 갖도록 한다

■ 문자열 상수나 문자열 배열을 초깃값으로 하는 생성자

다음과 같이 객체를 생성할 때 초깃값으로 문자열 상수를 지정하는 경우에 호출되는 생성자다.

```
MyString strA("Apple");
```

문자열 상수를 매개변수로 받아 객체를 생성해야 하므로 다음과 같이 매개변수가 char * 형태여야 한다. 매개변수로 넘겨진 값은 생성자 호출 후에 값이 변경되지 않도록 하려고 예약어 const를 붙였다. 매개변수로 넘겨진 문자열을 new 연산자로 새롭게 할당받은 기억공간에 저장해야 한다.

```
MyString::MyString(const char * const str)
{
  m_nLen = strlen(str)+1;      // -------------- ❶
  m_pStr = new char[m_nLen];  // -------------- ❷
  strcpy(m_pStr, str);         // -------------- ❸
}
```

❶ 우선 메모리를 할당할 기억공간의 크기를 알아내야 한다. 생성자의 매개변수인 str을 strlen 함수를 사용해서 문자열의 길이를 구한 후 여기에 더하기 1을 한 값을, 문자열의 길이를 저장하는 변수 m_nLen에 저장한다. 1을 더한 이유는 strlen 함수는 널 문자 전까지의 문자열의 길이를 재는데, 문자열을 저장할 공간은 널 문자가 저장될 공간까지 고려해서 메모리 할당을 해야 하기 때문이다.

❷ 할당받은 힙 영역의 주소값을 cahr *형 변수인 m_pStr에 저장한다.

❸ strcpy 함수를 사용해서 힙 영역에 새롭게 할당받은 메모리(m_pStr)에 매개변수로 넘겨준 문자열(str)을 복사한다.

■ **기본 생성자**

매개변수가 있는 생성자를 프로그래머가 정의하면 컴파일러로부터 제공받던 기본 생성자를 더 이상 제공받지 못하게 된다.

```
MyString strB;
```

그러므로 MyString 객체를 생성할 때 초깃값을 주지 않으려면 프로그래머가 기본 생성자를 반드시 정의해 주어야 한다. 널 문자를 저장하려고 문자열의 길이를 저장하는 변수 m_nLen에는 1을 대입한다. new 연산자로 1바이트의 메모리를 할당한 후에 m_pStr에 할당받은 기억공간의 주소를 저장한다. 할당된 기억공간에 널 문자를 저장한다.

```
MyString::MyString()
{
  m_nLen=1;
  m_pStr=new char[m_nLen];
  strcpy(m_pStr, "");
}
```

MyString 클래스 소멸자 정의하기

new에 의해 힙 영역에 잡힌 기억공간은 반드시 delete에 의해 메모리가 해제되어야 한다. 메모리 할당만 하고 해제를 하지 않으면 메모리 누수 현상에 의해서 접근 불가능한 메모리가 생겨 다른 자료를 저장할 수 없게 된다. 그리고 점차 사용할 수 있는 메모리 공간이 줄어서 조금만 큰 프로그램을 실행하면 금방 메모리가 부족해 메모리 할당을 할 수 없는 예외사항이 발생하게 된다.

MyString 클래스를 설계하는 과정에서도 생성자에서 new 연산자로 동적 메모리 할당을 하기 때문에 반드시 소멸자에서 할당된 메모리를 해제해 주어야 한다. 다음과 같이 하면 m_pStr이 가리키는 힙 영역을 delete 연산자로 메모리 해제하고 문자열의 길이는 0으로, m_pStr에는 NULL 포인터를 저장한다.

```
MyString::~MyString()
{
  delete []m_pStr;
  m_nLen = 0 ;
  m_pStr = NULL;
}
```

MyString 객체를 출력할 삽입 연산자(〈〈) 오버로딩하기

cout 객체를 선언한 ostream 클래스에는 문자형 포인터(char *)를 출력하도록 〈〈 연산자가 오버로딩되어 있다. 그러므로 문자열 상수, 문자열 배열, 문자열 포인터를 cout 객체의 〈〈 연산자를 통해서 출력할 수 있다. 하지만 우리가 새로 설계한 MyString 클래스에 대한 정보는 ostream 클래스에 명시되어 있지 않다. 그러므로 MyString 클래스를 설계하면서 〈〈 연산자에 대한 오버로딩도 함께 해주어야 MyString 클래스를 화면에 출력할 수 있다. 출력을 담당하는 삽입 연산자인 〈〈는 공식화되어 있기 때문에 이전에 학습한 연산자 오버로딩에서 Complex 객체를 〈〈 연산자의 오버로딩과 동일한 형태로 정의한다.

```
ostream & operator<<(ostream & os, MyString & temp)
{
  cout<<temp.m_pStr;
  return os;
}
```

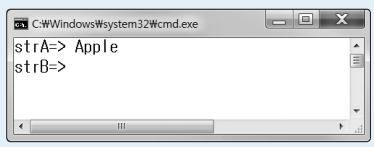
매개변수는 2개인데, 첫 번째 매개변수는 ostream &형, 두 번째 매개변수는 MyString &형이다. 출력 후에 〈〈 연산자를 연속해서 사용하려면 함수의 반환값도 ostream &이어야 한다. 이 때 〈〈 연산자는 MyString 클래스의 프렌드 함수로 선언한다. 왜냐하면 〈〈 연산자는 MyString 클래스의 멤버함수가 아니고 일반 함수지만 MyString 클래스의 private 멤버를 함수 내부에서 사용하기 때문이다.

```
friend ostream & operator<<(ostream &os, MyString & temp);
```

[예제 17-3]은 사용자가 정의한 생성자, 소멸자, 출력을 위해 〈〈 연산자만 이용해서 MyString 클래스를 설계한 첫 번째 프로그램이다.

예제 17-3 MyString 클래스 기본 설계하기(17_03.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08   int m_nLen;
```



```
09   char *m_pStr;
10 public :
11   MyString(const char * const str);
12   MyString();
13   ~MyString();
14   friend ostream & operator<<(ostream & os, MyString & temp);
15 };
16
17 MyString::MyString(const char * const str)
18 {
19   m_nLen = strlen(str)+1;
20   m_pStr = new char[m_nLen];
21   strcpy(m_pStr, str);
22 }
23
24 MyString::MyString()
25 {
26   m_nLen=1;
27   m_pStr=new char[m_nLen];
28   strcpy(m_pStr, "");
29 }
30
31 MyString::~MyString()
32 {
33   delete []m_pStr;
34   m_nLen = 0 ;
35   m_pStr = NULL;
36 }
37
38 ostream & operator<<(ostream & os, MyString & temp)
39 {
40   cout<<temp.m_pStr;
41   return os;
42 }
43
44 void main()
45 {
46   MyString strA("Apple");
47   MyString strB;
```

```
48
49    cout<<"strA=> "<< strA<<endl;
50    cout<<"strB=> "<< strB<<endl;
51
52    // MyString strC(strA);
53    // cout << "strC=> "<< strC << endl;
54  }
```

08행 MyString 클래스의 멤버변수는 2개인데, 하나는 문자열의 길이를 저장하기 위한 멤버변수로 정수형 m_nLen이다.

09행 다른 하나는 문자열을 저장할 힙 영역을 가리키는 멤버변수로, 문자형 포인터 m_pStr이다.

17행 문자열을 매개변수로 하는 생성자다. 46행처럼 문자열 상수나 문자 배열을 초깃값으로 설정해서 객체를 생성할 때 호출되는 생성자다. 매개변수로 전달된 문자열의 길이만큼 메모리를 할당한다.

19행 생성자의 매개변수인 str을 strlen 함수를 사용해서 문자열의 길이를 구한 후 여기에 1을 더해서 멤버변수 m_nLen에 저장한다.

20행 new 연산자를 통해 매개변수로 넘겨준 문자열을 저장할 수 있을 만큼의 기억공간을 할당받는다. 할당받은 힙 영역의 주소값을 cahr *형 변수인 m_pStr에 저장한다.

21행 힙 영역에 할당받은 메모리에 매개변수로 넘겨준 문자열(str)을 strcpy 함수를 사용해서 저장한다.

24행 매개변수가 없는 기본 생성자를 정의한다. 생성자를 하나도 정의하지 않으면 컴파일러가 기본 생성자를 제공해 주지만 일단 프로그래머가 생성자를 구현하면 컴파일러가 제공하는 기본 생성자를 사용할 수 없으므로 44행처럼 객체를 생성할 경우에 대비해서 기본 생성자도 일일이 작성해야 한다.

33행 기본 생성자는 new 연산자를 통해 동적으로 할당받은 기억공간을 delete로 해제한다.

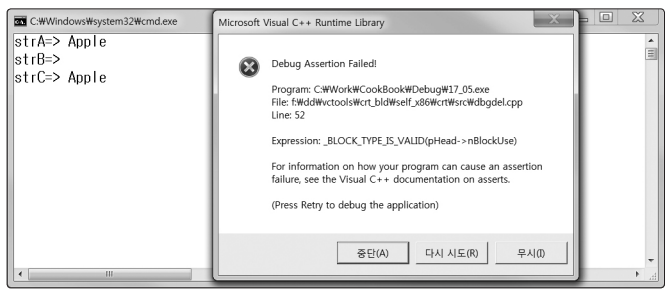
Section

03 복사 생성자

[예제 17-3]에서 52행~53행의 주석을 해제하여 실행해 보자.

```
52  MyString strC(strA);
53  cout<<"strC=> "<< strC<<endl;
```

객체 strC를 새롭게 생성하면서 이미 선언된 객체 strA를 초깃값으로 주었다. 프로그램이 잘 수행되다가 마지막 프로그램을 종료하기 직전에 다음과 같은 에러가 발생한다.



어떠한 문제가 있어서 이러한 예외사항이 발생하는 걸까? 그 이유부터 살펴본 후에 문제를 해결해 보자.

1. 기본 복사 생성자와 얕은 복사

기본 자료형인 int형 변수에 대해서 다음과 같이 변수 b는 생성되면서 변수 a를 초깃값으로 지정해 줄 수 있다. 그러면 새로 생성되는 변수 b는 변수 a의 값인 5를 초깃값으로 갖게 된다. 동일한 클래스로 선언된 2개의 객체들 사이에서 앞에서와 같은 정수형 변수 사이의 관계가 설정될 수 있다.

```
int a=5;
int b=a;
```

예를 들어 Complex 클래스로 객체를 생성할 경우 다음과 같이 프로그램을 작성할 수 있다. 객체 one은 초깃값으로 real, image 멤버변수에 100, 100을 갖는다. two는 one을 초깃값으로 주어 객체 two가 생성될 때 one의 두 멤버변수 값을 그대로 복사한다.

```
Complex one(100,100);
Complex two(one);
```

모든 객체는 객체 생성 시 초기화를 하려고 생성자가 호출된다. 객체 one을 생성할 때 호출되는 생성자는 (정수형으로 선언된) 매개변수 2개짜리 생성자고, 객체 two가 생성될 때는 자신과 동일한 클래스로 선언된 객체 one을 초깃값으로 주었기 때문에 기본 복사 생성자가 호출된다.

기본 복사 생성자는 매개변수 없는 기본 생성자처럼 C++ 컴파일러에 의해 제공되는 생성자인데, 객체 생성시 동일한 객체를 초깃값으로 줄 때 자동으로 호출된다. 이렇게 컴파일러에 의해서 제공되는 생성자는 클래스의 멤버변수들끼리 일대일 대응으로 값이 복사된다.

```
Complex two(one);
```

위와 같이 객체 생성을 할 경우 내부적으로 다음과 같은 동작이 일어나는데, 이렇게 컴파일러에 의해서 제공되는 기본 복사 생성자에 의해서 멤버변수 값이 복사되는 것을 '얕은 복사(shallow copy)'라고 한다.

```
two.real = one.real;
two.image = one.image;
```

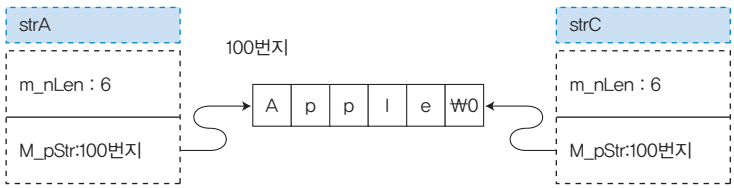
2. 복사 생성자의 오버로딩과 깊은 복사

컴파일러에 의해 제공되는 복사 생성자가 특정 클래스에 대해서는 문제를 일으킨다. 기본 복사 생성자를 사용하면 안 되는 대표적인 예가 바로 생성자에서 동적 메모리 할당을 하는 클래스의 경우다.

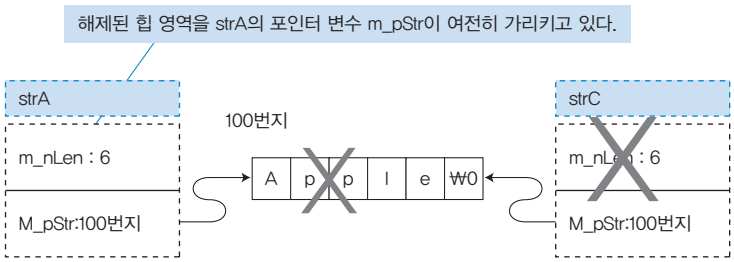
```
MyString strA("Apple");
```

객체 strA가 생성될 때 힙 영역에 동적 메모리를 할당하고 그 기억공간을 멤버변수 m_pStr이 가리키고 있다. 객체 strA를 초깃값으로 하여 strC를 생성하게 되면 기본 복사 생성자에 의해서 얕은 복사가 일어나서 멤버변수끼리 값이 복사되어 자료값을 저장하고 있는 힙 영역을 두 객체가 가리키는 구조가 된다.

```
MyString strC(strA);
```



strA와 strB가 서로 동일한 힙 영역을 가리키고 있다가 두 객체가 소멸될 때 소멸자가 각각 호출되면서 이미 메모리 해제된 힙 영역을 다시 메모리 해제하려고 하는 문제가 발생한다.



나중에 선언된 객체가 먼저 소멸되므로 객체 strC가 먼저 소멸된다. 객체 strC가 소멸되면서 이미 힙 영역의 자료를 저장하는 공간은 사라졌다. 하지만 객체 strA의 m_pStr이 여전히 힙 영역을 가리키고 있다가 strA가 소멸될 때 다시 한 번 메모리 해제를 시도하게 된다.

이러한 문제를 해결하려면 복사 생성자가 단순히 멤버변수만 복사(얕은 복사)하는 것이 아니라 새로 생성되는 객체가 별개의 메모리를 힙 영역에 할당받도록 프로그래머가 직접 정의해야 한다. 우선 C++컴파일러에 의해 제공되는 기본 생성자의 기본 형태를 살펴보자.

기본 생성자 기본 형식

```
클래스명(const 클래스명 &객체명);
```

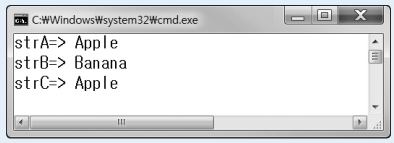
복사 생성자도 생성자이므로 함수명이 클래스명과 동일해야 하고, 복사 생성자 호출 시 이미 존재하는 객체를 실 매개변수로 전달해주므로 이를 형식 매개변수가 참조 변수 형태로 받아서 처리한다.

참조란 이미 선언된 객체에 별칭을 붙여주는 것이므로 따로 메모리 할당을 하지 않는다. 만일 복사 생성자의 형식 매개변수를 값에 의한 호출 방식으로 지정한다면 따로 메모리 할당을 하면서 얕은 복사를 하게 되므로 반드시 참조에 의한 호출 방식으로 지정해야 한다. 또한 생성자를 호출한 후에 원래 존재하던 객체의 값이 변경되어서는 안 되기 때문에 객체명 앞에 예약어 const를 덧붙이다. const는 상수라는 의미를 내포하고 있는 지정어로 별칭이 가리키는 객체의 값을 가져다 사용만 할 뿐, 값을 변경하지 못하게 하기 위해 사용한다. 그럼 직접 복사 생성자를 만들어 보자.

기본 복사 생성자는 얕은 복사를 하므로 깊은 복사를 하게 하려고 복사 생성자를 정의하는 프로그램이 [예제 17-4]다.

예제 17-4 깊은 복사를 위한 복사 생성자 정의하기(17_04.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08   int m_nLen;
09   char *m_pStr;
10 public :
11   MyString();
12   MyString(const char * const str);
13   ~MyString();
14   friend ostream & operator<<(ostream & os, MyString & temp);
15   MyString(const MyString& str);
16 };
17
```

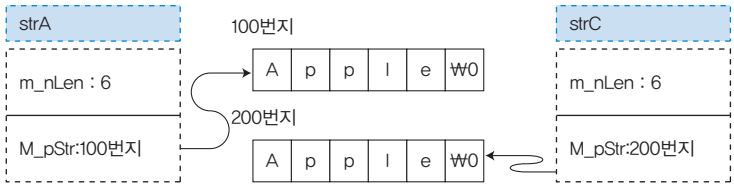


```
18 MyString::MyString(const MyString & src)
19 {
20   m_nLen=src.m_nLen;
21   m_pStr=new char[m_nLen];
22   strcpy(m_pStr, src.m_pStr);
23 }
24
25 MyString::MyString(const char * const str)
26 {
27   m_nLen = strlen(str)+1;
28   m_pStr = new char[m_nLen];
29   strcpy(m_pStr, str);
30 }
31
32 MyString::MyString()
33 {
34   m_nLen=1;
35   m_pStr=new char[m_nLen];
36   strcpy(m_pStr, "");
37 }
38
39 MyString::~MyString()
40 {
41   delete []m_pStr;
42   m_nLen = 0 ;
43   m_pStr = NULL;
44 }
45
46 ostream & operator<<(ostream & os, MyString & temp)
47 {
48   cout<<temp.m_pStr;
49   return os;
50 }
51
52 void main()
53 {
54   MyString strA("Apple");
55   MyString strB("Banana");
56
```

```
57   cout<<"strA=> "<< strA<<endl;
58   cout<<"strB=> "<< strB<<endl;
59
60   MyString strC(strA);
61   cout<<"strC=> "<< strC<<endl;
62
63   // strB=strA;
64   // cout<<"strB=> "<< strB<<endl;
65 }
```

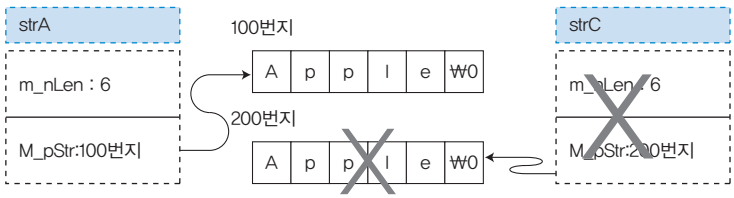
15행 클래스 MyString에 복사 생성자의 원형을 정의했다.

60행 객체 strA를 초깃값으로 하여 strC를 생성하게 되면 18행~23행에서 정의한 복사 생성자가 호출된다.



이번에는 21행에서 힙 영역에 배열을 위한 기억공간을 따로 할당한다. 22행에서 따로 할당된 기억공간에 문자열 상수를 저장함으로서 깊은 복사를 할 수 있도록 했다.

strA와 strC의 포인터 변수 m_pStr이 서로 다른 배열을 가리키므로 객체 strA가 소멸될 때 해제하는 힙 영역과 객체 strC가 소멸될 때 해제하는 힙 영역이 서로 다르므로 에러가 발생하지 않는다.



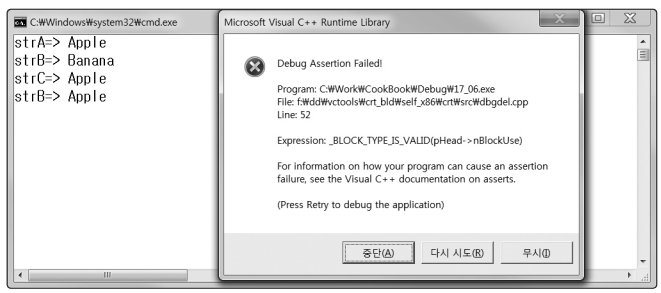
Section

04 대입 연산자 오버로딩

[예제 17-4]의 63행~64행의 주석을 풀고 실행해 보자.

```
63  strB=strA;
64  cout<<"strB=> "<< strB<<endl;
```

이미 생성된 객체 strA의 값을 strB에 대입했다. 프로그램이 잘 수행되는 듯 하다가 마지막 프로그램을 종료하기 바로 전에 다음과 같은 에러가 발생한다. 그것도 실행하는 중에 에러(예외사항)가 발생한다.



어떠한 문제가 있기에 이러한 예외사항이 발생하는 걸까? 그 이유부터 살펴본 후에 문제를 해결해 보자.

1 기본 대입 연산자와 얕은 복사

int형 변수에 대해서 다음과 같이 변수 b의 값을 변수 a에 대입해 줄 수 있다. 그러면 변수 b에 변수 a의 값인 5가 저장된다.

```
int a=5;
int b=10;
b=a;
```

동일한 클래스로 선언된 2개의 객체들 사이에서 위와 같은 정수형 변수 사이의 관계가 설정될 수 있다. 예를 들어 Complex 클래스로 객체를 생성할 경우 다음과 같이 프로그램을 작성할 수 있다. 객체 one은 초깃값으로 real, image 멤버변수에 100, 100을 갖는다. two에 one을 대입하면 객체 two에 객체 one의 두 멤버변수 값을 그대로 복사한다.

```
Complex one(100,100);
Complex two(50, 50);
two=one;
```

이렇게 객체끼리 값을 대입하는 것은 서로 다른 클래스로 선언된 객체끼리는 불가능하고 동일한 클래스로 선언된 객체끼리만 가능한데, 이는 C++ 컴파일러가 기본 대입 연산자를 제공해 주기 때문이다. 이렇게 컴파일러에 의해서 제공되는 대입 연산자는 클래스의 멤버변수들끼리 일대일 대응으로 값이 복사된다. 내부적으로 일어나는 동작은 다음과 같은데, 이렇게 컴파일러에 의해서 제공되는 기본 대입 연산자에 의해서 멤버변수 값이 복사되는 것을 '얕은 복사(shallow copy)'라고 한다.

```
two.real = one.real;
two.image = one.image;
```

2. 대입 연산자 오버로딩과 깊은 복사

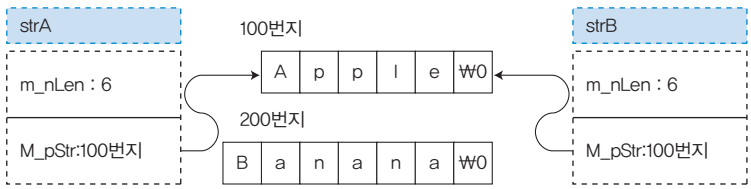
컴파일러에 의해 제공되는 대입 연산자가 특정 클래스에 대해서는 문제를 일으킨다. 기본 대입 연산자를 사용하면 안 되는 대표적인 예가 바로 생성자에서 동적 메모리 할당을 하는 클래스의 경우다.

```
MyString strA("Apple");
MyString strB("Banana");
```

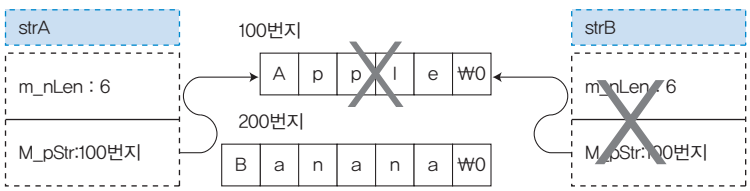
객체 strA와 객체 strB가 생성될 때 힙 영역에 동적 메모리 할당을 하고 그 기억공간을 멤버변수 m_pStr이 가리키고 있다. 객체 strA는 "Apple"을 저장하려고 힙 영역에 6바이트 메모리 할당을 했고, 객체 strB는 "Banana"를 저장할 수 있도록 힙 영역에 따로 7바이트 메모리 할당을 했다.

다음 예처럼 strA를 strB에 저장한다고 해 보자.

```
strB=strA;
```

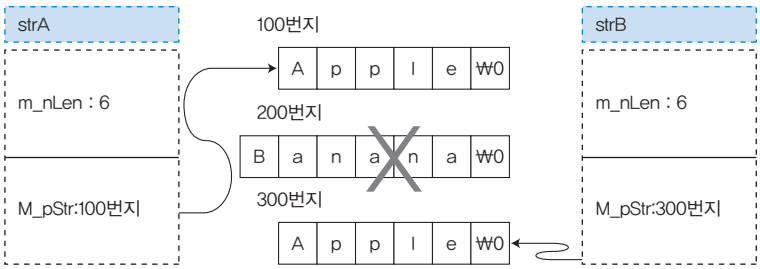


MyString 클래스의 멤버값 2개만이 복사되는 얕은 복사로 인해 strA가 가리키는 힙 역을 strB가 가리키게 된다. 이렇게 두 객체가 동일한 힙 영역을 가리키게 되면, 객체가 소멸될 때 소멸자에 의해서 소멸된 기억공간을 다시 소멸하려고 시도하기 때문에 디버깅 에러가 발생한다.



strB는 strA의 내용만 복사하지 않고, strA와 동일한 형태의 기억공간을 별도로 할당받는 깊은 복사가 이루어져야 별 문제없이 두 객체 사이에 대입 연산자를 사용할 수 있게 된다.

다음은 객체값을 대입 연산자로 치환했을 때 바람직한 형태를 그린 그림이다.



MyString 클래스에서 대입 연산자가 문제를 일으키지 않으려면 포인터 변수 m_pStr이 대입 연산자 오른쪽 피연산자의 m_pStr 영역을 가리키는 것이 아니고, 다음과 같이 동작해야 한다.

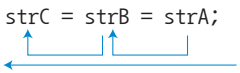
❶ 왼쪽 피연산자가 가리키는 힙 영역을 메모리 해제한다.

❷ 오른쪽 피연산자로 사용된 객체와 동일한 크기의 메모리를 별개의 기억공간으로 따로 할당받는다.

❸ 그 곳에 왼쪽 피연산자가 가지고 있는 자료값을 복사해야 한다. 위에 언급한 사항을 내용으로 하는 대입 연산자는 다음과 같이 정의한다.

```
MyString& MyString::operator=(const MyString &temp)
{
  if(this == &temp)
    return *this;
  delete [] m_pStr;
  m_nLen = temp.m_nLen;
  m_pStr = new char[m_nLen];
  strcpy(m_pStr, temp.m_pStr);
  return *this;
}
```

동적 메모리를 할당하는 MyString 객체는 = 연산자가 멤버변수 값만 그대로 대입하는 기본 대입 연산자를 사용해서는 안 되고 복사 생성자를 재정의했듯이 = 연산자도 재정의해야 한다. = 연산자를 기준으로 오른쪽에 있는 객체의 크기와 동일한 크기로 왼쪽에 있는 객체를 메모리에 새로 할당하여 오른쪽 객체에 저장된 문자열을 복사해야 한다. = 연산자도 << 연산자처럼 연속적으로 사용해야 하므로 반환값이 MyString &이어야 한다.



대입 연산자는 strA에 저장된 값을 strB에 저장하고 대입 연산한 결과값을 다시 strC에 저장한다. = 연산자를 MyString 클래스의 멤버함수로 구현했다면 컴파일러에 의해 호출되는 형태는 다음과 같다.

```
strC.operator=(strB.operator=(strA))
```

strB 객체와 strA 객체를 실 매개변수로 하여 operator= 연산자 함수를 호출한 결과값을 다시 operator= 연산자의 매개변수로 사용해야 하므로 operator= 연산자 함수의 결과값은 MyString&이어야 한다. 그리고 = 연산자는 이항 연산자이므로 피연산자 2개가 있어야 하는데 = 연산자를 MyString 클래스의 멤버함수로 구현하게 되면 피연산자 2개 중 왼쪽 피연산자는 this에 의해 관리되므로 매개변수는 한 개만 기술하면 된다.

```
MyString& MyString::operator=(const MyString & temp)
{
  ...
}
```

= 연산자 함수의 원형 정의는 대략 형태가 나왔다. 이번에는 함수를 어떻게 구현해야 할지 살펴보자. 연산자를 기준으로 왼쪽 객체는 이전에 할당받은 힙 영역을 해제하고 오른쪽 객체와 동일한 크기의 메모리를 다시 할당받아야 한다.

```
strA=strA;
```

하지만 만일 위와 같이 대입 연산자가 사용되었다면 동일한 객체를 대입하게 되어 내부적으로 아무런 일도 일어나지 않는다. 그러므로 대입 연산자 오른쪽 피연산자와 왼쪽 피연산자가 동일 객체인지 살펴보고 동일 객체라면 아무런 작업도 하지 않고 함수를 종료한다. 대입 연산자 왼쪽 객체의 주소값이 this에 의해서 관리되므로 이 주소와 대입 연산자 오른쪽 객체가 매개변수로 전달되고 매개변수의 주소값이 동일한 객체라면 두 객체가 같다고 보고 대입 연산자 왼쪽 객체를 결과값으로 반환하고 여기서 함수를 종료시킨다. = 연산자 함수의 반환값이 MyString이므로 왼쪽 피연산자에 해당되는 객체값을 결과값으로 반환해야 하므로 return문 다음에 *this를 기술했다.

```
if(this == &temp)
  return *this;
```

동일한 객체를 대입하는 것이 아니라면 this로 참조되는 객체의 메모리를 해제하고 매개변수에 의해 관리되는 객체의 m_nLen 멤버변수의 크기로 메모리를 재할당받아야 한다.

```
delete [] m_pStr;
m_nLen = temp.m_nLen;
m_pStr = new char[m_nLen];
```

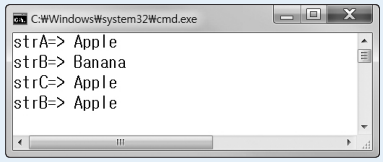
메모리 할당 후에는 대입 연산자 오른쪽 객체가 저장하고 있는 문자열 상수를 대입 연산자 왼쪽 객체에 복사한 후 왼쪽 피연산자가 동일 객체인지 살펴보고 동일 객체라면 아무런 작업도 하지 않고 함수를 종료한다. 연산자 함수의 반환값이 MyString이므로 왼쪽 피연산자에 해당되는 객체값을 결과값으로 반환하므로 return문 다음에 *this를 기술했다.

```
strcpy(m_pStr, temp.m_pStr);
return *this;
```

기본 대입 연산자는 얕은 복사를 하므로 깊은 복사를 하려고 대입 연산자를 오버로딩하는 프로그램이 [예제 17-5]다.

예제 17-5 깊은 복사를 위한 대입 연산자 오버로딩하기(17_05.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08   int m_nLen;
09   char *m_pStr;
10 public :
11   MyString();
12   MyString(const char * const str);
13   ~MyString();
14   friend ostream & operator<<(ostream & os, MyString & temp);
15   MyString(const MyString& str);
16   MyString& operator=(const MyString &temp);
```



```
17 };
18
19 MyString& MyString::operator=(const MyString &temp)
20 {
21   if(this == &temp)
22     return *this;
23
24   delete [] m_pStr;
25   m_nLen = temp.m_nLen;
26   m_pStr = new char[m_nLen];
27   strcpy(m_pStr, temp.m_pStr);
28   return *this;
29 }
30 // 17_06.cpp의 내용과 중복
31 MyString::MyString(const MyString & src)
32 {
33   m_nLen=src.m_nLen;
34   m_pStr=new char[m_nLen];
35   strcpy(m_pStr, src.m_pStr);
36 }
37
38 MyString::MyString(const char * const str)
39 {
40   m_nLen = strlen(str)+1;
41   m_pStr = new char[m_nLen];
42   strcpy(m_pStr, str);
43 }
44
45 MyString::MyString()
46 {
47   m_nLen=1;
48   m_pStr=new char[m_nLen];
49   strcpy(m_pStr, "");
50 }
51
52 MyString::~MyString()
53 {
54   delete []m_pStr;
55   m_nLen = 0 ;
```

```
56   m_pStr = NULL;
57 }
58
59 ostream & operator<<(ostream & os, MyString & temp)
60 {
61   cout<<temp.m_pStr;
62   return os;
63 }
64
65 void main()
66 {
67   MyString strA("Apple");
68   MyString strB("Banana");
69
70     cout<<"strA=> "<< strA<<endl;
71     cout<<"strB=> "<< strB<<endl;
72
73     MyString strC(strA);
74     cout<<"strC=> "<< strC<<endl;
75
76     strB=strA;
77     cout<<"strB=> "<< strB<<endl;
78
79     // strA="Apple";
80     // strB="Banana";
81     // strC=strA+strB;
82     // cout<<"strA=> "<< strA<<endl;
83     // cout<<"strB=> "<< strB<<endl;
84     // cout<<"strC=> "<< strC<<endl;
85
86     // cout<<"strC[2]=>"<<strC[2]<<endl;
87 }
```

21행~22행 원본과 대상이 동일한 객체라면 자기 자신을 반환한다.

24행 왼쪽 피연산자가 가리키는 힙 영역을 메모리 해제한다.

26행 오른쪽 피연산자와 동일한 크기의 메모리를 따로 할당받는다.

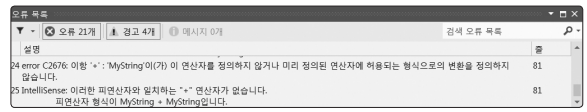
27행 따로 할당받은 메모리에 왼쪽 피연산자가 가지고 있는 문자열을 복사해야 한다.

Section

05 + 연산자 오버로딩

[예제 17-5]의 79행~84행의 주석 처리를 풀고 실행해 보자. + 연산자를 사용해서 문자열 2개를 연결하려고 하면 다음과 같이 컴파일 에러가 발생한다.

```
79  strA = "Apple";
80  strB = "Banana";
81  strC = strA + strB;
82  cout<<"strA=> "<< strA<<endl;
83  cout<<"strB=> "<< strB<<endl;
84  cout<<"strC=> "<< strC<<endl;
```



MyString 클래스에 + 연산자를 오버로딩한 후에야 위와 같이 기술해서 두 문자열을 결합할 수 있게 된다. + 연산자를 오버로딩하기 전에 + 연산자를 멤버함수로 구현할 경우 + 연산자가 컴파일러에 의해 어떻게 호출되는지 분석해 보자.

```
strA.operator+(strB);  // strA + strB;
```

operator+ 연산자 함수를 호출하는 객체 strA는 this에 의해 관리되므로 strB만 매개변수로 기술하면 된다. operator+ 연산자 함수가 호출되는 형태를 기준으로 연산자 함수의 원형을 정의해보면 다음과 같다.

```
MyString& MyString::operator+(const MyString & rightHand)
{
  ...
}
```

+ 연산자는 MyString 클래스 2개를 피연산자로 하고 그 결과로 두 문자열을 결합한다. + 연산자는 두 피연산자의 값은 그대로 두고 연결된 문자열이 결과값으로 구해져야 한다. + 연산자를 수행하고 난 후 strA에는 “Apple”이, strB에는 “Banana”가, 그리고 strC에는 “AppleBanana”가 저장되어 있어야 한다. C++에서 두 문자열을 연결해 주는 strcat 함수와는 동작 원리가 다르다.

```
MyString MyString::operator +(const MyString &rightHand)
{
  int tot_len = m_nLen + rightHand.m_nLen-1;
  char * temp = new char[tot_len];
  strcpy(temp, m_pStr);
  strcat(temp, rightHand.m_pStr);
  MyString result(temp);
  delete [] temp;
  return result;
}
```

+ 연산자 함수에서는 두 문자열의 길이를 더한 크기로 임시 객체를 만들어 주어야 한다. 우선 두 문자열의 길이를 구한다. 지역 변수 tot_len에는 왼쪽 피연산자의 문자열의 길이와 오른쪽 피연산자의 문자열 길이를 더한 값에서 1을 뺀다. m_nLen은 널 문자까지를 포함한 문자열의 길이이므로 왼쪽 피연산자의 길이에 오른쪽 피연산자의 문자열의 길이를 더하게 되면 널 문자를 위한 기억공간의 값이 2번 더해지는 셈이 되므로 1을 뺀 것이다.

```
int tot_len = m_nLen + rightHand.m_nLen-1;
```

두 문자열을 모두 저장할 만한 크기의 문자 배열을 생성한다.

```
char * temp = new char[tot_len];
```

문자 배열에 + 연산자를 기준으로 왼쪽 객체의 문자열을 우선 저장하고 오른쪽 객체를 연결시킨다. MyString 객체를 새로 생성하여 두 문자열을 결합한 문자 배열을 초깃값으로 준다.

```
strcpy(temp, m_pStr);
strcat(temp, rightHand.m_pStr);
```

문자 배열을 초깃값으로 하는 새로운 MyString 객체 result를 생성한 후 temp는 메모리를 해제하고 두 문자열이 연결되어 저장되어 있는 result 객체를 연산자 함수의 반환값으로 반환한다.

```
MyString result(temp);
delete [] temp;
return result;
```

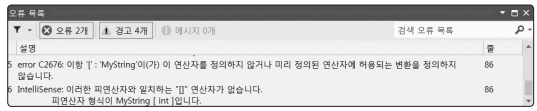
Section

6 [ ] 연산자 오버로딩

문자열 배열의 경우 원하는 위치값을 배열명 다음에 오는 대괄호([ ])안에 기술하면 그 위치의 문자값 하나를 알아낼 수 있다. 하지만 클래스로 선언된 객체 다음에 대괄호를 기술하고 그 안에 정수값을 기술하면 컴파일 에러가 발생한다.

[예제 17-5]에서 86행의 주석문을 풀고 실행해 보자.

```
86  cout<<"strC[2]=>"<<strC[2]<<endl;
```



[ ] 연산자를 사용해서 MyString 클래스에 저장되어 있는 문자열 중 특정 위치의 문자 하나만 알아내려고 [ ] 연산자를 오버로딩해야 한다. 우선 [ ] 연산자를 오버로딩하기 전에 [ ] 연산자를 멤버함수로 구현할 경우 [ ] 연산자가 컴파일러에 의해 어떻게 호출되는지 분석해 보자.

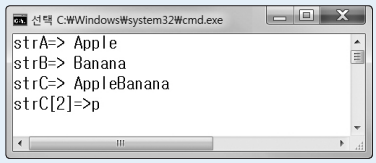
```
strC.operator[](2) // strC[2]
```

[ ] 연산자도 이항 연산자다. [ ] 연산자를 기준으로 왼쪽에는 MyString 클래스가 오고 오른쪽에는 정수값이 기술된다. 첨자 연산자도 멤버함수로 기술하면 왼쪽 MyString 클래스는 this에 의해 관리되고 오른쪽 정수값은 매개변수에 의해 관리된다. 매개변수로 전달된 첨자가 문자열의 길이보다 크다면 마지막 문자를 반환하고 아니면 첨자 위치의 문자 한 개를 반환한다. [ ] 연산자의 반환값은 char형이다.

```
char MyString::operator[](int n)
{
  if( n > m_nLen )
  return m_pStr[m_nLen-1];
  else
  return m_pStr[n];
}
```

[예제 17-6]은 두 문자열을 결합하기 위한 + 연산자와 첨자를 지정해서 해당 위치의 문자를 얻어내는 [ ] 연산자를 오버로딩하는 프로그램이다.

예제 17-6 연산자(+, [ ]) 오버로딩하기(17_06.cpp)




```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08   int m_nLen;
09   char *m_pStr;
10 public :
11   MyString();
12   MyString(const char * const str);
13   ~MyString();
14   friend ostream & operator<<(ostream & os, MyString & temp);
15   MyString(const MyString& str);
16   MyString& operator=(const MyString &temp);
17   MyString operator +(const MyString &rightHand);
18   char operator[](int n);
19 };
20
21 MyString MyString::operator +(const MyString &rightHand)
22 {
23   int tot_len = m_nLen + rightHand.m_nLen-1;
24   char * temp = new char[tot_len];
25   strcpy(temp, m_pStr);
26   strcat(temp, rightHand.m_pStr);
27   MyString result(temp);
28   delete [] temp;
29   return result;
30 }
31
32 char MyString::operator[](int n)
33 {
34   if( n > m_nLen )
35     return m_pStr[m_nLen-1];
36   else
```

```
37    return m_pStr[n];
38 }
39
40 MyString& MyString::operator=(const MyString &temp)
41 {
42   if(this == &temp)
43     return *this;
44
45   delete [] m_pStr;
46   m_nLen = temp.m_nLen;
47   m_pStr = new char[m_nLen];
48   strcpy(m_pStr, temp.m_pStr);
49   return *this;
50 }
51
52 MyString::MyString(const MyString & src)
53 {
54   m_nLen=src.m_nLen;
55   m_pStr=new char[m_nLen];
56   strcpy(m_pStr, src.m_pStr);
57 }
58
59 MyString::MyString()
60 {
61   m_nLen=1;
62   m_pStr=new char[m_nLen];
63   strcpy(m_pStr, "");
64 }
65
66 MyString::MyString(const char * const str)
67 {
68   m_nLen = strlen(str)+1;
69   m_pStr = new char[m_nLen];
70   strcpy(m_pStr, str);
71 }
72
73 MyString::~MyString()
74 {
75   delete []m_pStr;
76   m_nLen = 0 ;
77   m_pStr = NULL;
```

```
78 }
79
80 ostream & operator<<(ostream & os, MyString & temp)
81 {
82   cout<<temp.m_pStr;
83   return os;
84 }
85
86 void main()
87 {
88   MyString strA("Apple");
89   MyString strB("Banana");
90   MyString strC;
91
92   strC=strA+strB;
93   cout<<"strA=> "<<strA<<endl;
94   cout<<"strB=> "<<strB<<endl;
95   cout<<"strC=> "<<strC<<endl;
96
97   cout<<"strC[2]=>"<<strC[2]<<endl;
98  }
```

92행 이 문장을 컴파일러가 해석하는 식으로 변경해 보면 다음과 같다.

```
strC.operator=( strA.operator+(strB) );
```

연산자 함수를 호출하는 형태로 변경해보면 + 연산자와 = 연산자가 호출되는 것을 확실하게 알 수 있다. 연산자 우선순위에 의해서 strA.operator+(strB)가 먼저 수행된다. MyString 객체에 멤버함수로 재정의한 23행의 덧셈 연산자(+)가 호출된다.

23행~26행 덧셈 연산자(+)는 두 문자열을 연결하여 저장하려고 임시 기억장소 temp를 두 문자열 길이의 합을 저장하고 있는 tot_len 크기만큼 할당한다. temp에 왼쪽 객체(strA)의 문자열을 저장하고 오른쪽 객체(strB)의 문자열은 추가적으로 연결한다.

27행 MyString 클래스로 새로운 객체 result를 생성하면서 초깃값으로 문자 배열 temp를 지정한 후 28행에서 temp를 메모리 해제한다.

29행 두 문자열을 연결하여 저장하고 있는 객체 result의 값을 반환한다.

97행 strC는 MyString 클래스로 선언된 객체다. MyString 클래스에 [ ] 연산자를 재정의했기 때문에 특정 첨자 위치의 문자 값을 알아낼 수 있다. strC[2]는 strC.operator[ ](2)와 같은 의미이므로 35행에 재정의된 [ ] 연산자가 호출된다.

34행~37행 첨자가 문자열의 길이보다 크지 않으므로 37행의 else 다음의 문장인 return m_pStr[n];이 수행된다. m_pStr 배열의 원소값은 char형이므로 operator[ ] 연산자 함수의 자료형이 char다.

Section

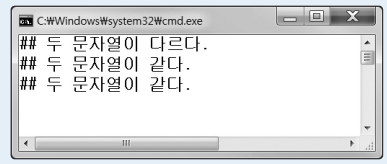
07 관계 연산자 오버로딩

두 문자열을 비교하려면 함수를 사용해서 문자 배열을 만들어야 한다. 하지만 C++에서 제공해 주는 string 클래스는 관계 연산자로 문자열을 비교할 수 있다. MyString 클래스에 관계 연산자를 적용해보기 전에 먼저 C++의 string 클래스에서 제공하는 관계 연산자의 동작 원리부터 예제를 통해 알아보자.

[예제 17-7]은 C++의 string 클래스가 제공하는 두 문자열 값이 같은지를 알려주는 == 관계 연산자의 동작 원리를 살펴보기 위한 프로그램이다.

예제 17-7 string 클래스의 관계 연산자 동작 원리 알아보기(17_07.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 void main()
06 {
07   string strA("Apple");
08   string strB("Banana");
09
10   if(strA == strB)
11     cout<<"## 두 문자열이 같다.\n";
12   else
13     cout<<"## 두 문자열이 다르다.\n";
14
15   strB="Apple";
16
17   if(strA == strB)
18     cout<<"## 두 문자열이 같다.\n";
19   else
20     cout<<"## 두 문자열이 다르다.\n";
21
```



```
22   if(strA == "Apple")
23     cout<<"## 두 문자열이 같다.\n";
24   else
25     cout<<"## 두 문자열이 다르다.\n";
26 }
```

10행 string 클래스로 선언된 객체끼리 서로 동일한 문자열이 저장되어 있는지를 == 연산자로 살펴봤다. 두 객체 strA와 strB에 저장된 문자열이 다르므로 == 연산자는 false를 반환하므로 12행에 else 다음에 있는 문장인 "두 문자열이 다르다."가 출력된다.

15행 strB에 "Apple"을 저장한 후에 17행에서 다시 strA와 strB의 값이 같은지를 물어보면 이번에는 == 연산자가 true를 반환하므로 17행의 if문 다음에 기술한 문장인 "두 문자열이 같다."가 출력된다.

22행 이번에는 string 객체와 문자열 상수와의 대소 관계를 == 연산자로 물어보았다. 두 문자열 값은 같으므로 == 연산자는 true를 반환하여 22행의 if문 바로 다음에 기술한 "두 문자열이 같다."가 출력된다.

이제 이 관계 연산자를 MyString 클래스에 직접 적용해 보자. [예제 17-8]은 == 연산자를 MyString 클래스에 적용해보는 프로그램이다.

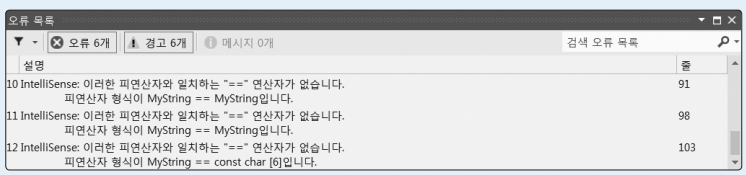
예제 17-8 MyString 클래스에 == 관계 연산자 사용하기(17_08.cpp)

```
001 #include<iostream>
002 #include<string>
003 using namespace std;
004
005 class MyString
006 {
007 private :
008   int m_nLen;
009   char *m_pStr;
010 public :
011   MyString();
012   MyString(const char * const str);
013   ~MyString();
014   friend ostream & operator<<(ostream & os, MyString & temp);
015   MyString(const MyString& str);
016   MyString& operator=(const MyString &temp);
017   MyString operator+(const MyString &rightHand);
018   char operator[](int n);
019 };
020
```

```
021 MyString MyString::operator+(const MyString &rightHand)
022 {
023   int tot_len = m_nLen + rightHand.m_nLen-1;
024   char * temp = new char[tot_len];
025  strcpy(temp, m_pStr);
026   strcat(temp, rightHand.m_pStr);
027   MyString result(temp);
028   delete [] temp;
029   return result;
030 }
031
032 char MyString::operator[](int n)
033 {
034   if( n > m_nLen )
035     return m_pStr[m_nLen-1];
036   else
037     return m_pStr[n];
038 }
039
040 MyString& MyString::operator=(const MyString &temp)
041 {
042   if(this == &temp)
043     return *this
044
045   delete [] m_pStr;
046   m_nLen = temp.m_nLen;
047   m_pStr = new char[m_nLen];
048   strcpy(m_pStr, temp.m_pStr);
049   return *this
050 }
051
052 MyString::MyString(const MyString & src)
053 {
054   m_nLen=src.m_nLen;
055   m_pStr=new char[m_nLen];
056   strcpy(m_pStr, src.m_pStr);
057 }
058
059 MyString::MyString()
```

```
060 {
061   m_nLen=1;
062   m_pStr=new char[m_nLen];
063   strcpy(m_pStr, "");
064 }
065
066 MyString::MyString(const char * const str)
067 {
068   m_nLen = strlen(str)+1;
069   m_pStr = new char[m_nLen];
070   strcpy(m_pStr, str);
071 }
072
073 MyString::~MyString()
074 {
075   delete []m_pStr;
076   m_nLen = 0
077   m_pStr = NULL
078 }
079
080 ostream & operator<<(ostream & os, MyString & temp)
081 {
082   cout<<temp.m_pStr;
083   return os
084 }
085
086 void main()
087 {
088   MyString strA("Apple");
089   MyString strB("Banana");
090
091   if(strA == strB)
092     cout<<"## 두 문자열이 같다.\n";
093   else
094     cout<<"## 두 문자열이 다르다.\n";
095
096   strB="Apple";
097
098   if(strA == strB)
```

```
099     cout<<"## 두 문자열이 같다.\n";
100   else
101     cout<<"## 두 문자열이 다르다.\n";
102
103   if(strA == "Apple")
104     cout<<"## 두 문자열이 같다.\n";
105   else
106     cout<<"## 두 문자열이 다르다.\n";
107 }
```



91행, 98행, 103행 MyString 클래스에 == 연산자를 오버로딩하지 않고 MyString 객체에 사용하면 컴파일 에러가 발생한다. 그러므로 MyString 클래스에 == 연산자를 오버로딩해야만 한다.

다음은 MyString 클래스에 == 연산자를 오버로딩한 것이다.

```
bool MyString::operator==(const MyString &leftHand)
{
  if(strcmp(m_pStr, leftHand.m_pStr)==0)
  return true;
  else
  return false;
}
```

strcmp 함수는 문자열의 내용을 비교하는 함수다. strcmp 함수의 결과가 0이면 두 문자열의 내용이 같은 것이므로 true를 반환한다. strcmp 함수의 첫 번째 매개변수가 두 번째 매개변수보다 크면 양수를 반환하고 작으면 음수를 반환한다. strcmp 함수의 결과가 0이 아니면 무조건 false를 반환한다. 만약 MyString 객체끼리 비교하는 것이 아니고 문자열 상수와 비교할 경우에는 위에 정의한 == 연산자를 사용할 수 없다. 문자열 상수를 형식 매개변수로 갖는 == 연산자를 재정의해야 한다.

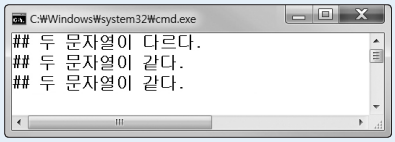
```
bool MyString::operator==(const char *leftHand)
{
  if(strcmp(m_pStr, leftHand)==0)
```

```
  return true;
  else
  return false;
}
```

[예제 17-9]는 MyString 클래스에 두 문자열이 같은지 판단하는 관계 연산자를 구현한 프로그램이다.

예제 17-9 두 문자열이 같은지 판단하는 관계 연산자 구현하기(17_09.cpp)

```
001 #include<iostream>
002 #include<string>
003 using namespace std;
004
005 class MyString
006 {
007 private :
008   int m_nLen;
009   char *m_pStr;
010 public :
011   MyString();
012   MyString(const char * const str);
013   ~MyString();
014   friend ostream & operator<<(ostream & os, MyString & temp);
015   MyString(const MyString& str);
016   MyString& operator=(const MyString &temp);
017   MyString operator +(const MyString &rightHand);
018   char operator[](int n);
019   bool operator==(const MyString &leftHand);
020   bool operator==(const char *leftHand);
021 };
022
023 bool MyString::operator==(const MyString &leftHand)
024 {
025   if(strcmp(m_pStr, leftHand.m_pStr)==0)
026   return true;
027   else
028   return false;
029 }
```



```
030
031 bool MyString::operator==(const char *leftHand)
032 {
033   if(strcmp(m_pStr, leftHand)==0)
034     return true;
035   else
036     return false;
037 }
038 // 17_10.cpp의 내용과 중복
039 MyString MyString::operator+(const MyString &rightHand)
040 {
041   int tot_len = m_nLen + rightHand.m_nLen-1;
042   char * temp = new char[tot_len];
043  strcpy(temp, m_pStr);
044   strcat(temp, rightHand.m_pStr);
045   MyString result(temp);
046   delete [] temp;
047   return result;
048 }
049
050 char MyString::operator[](int n)
051 {
052   if( n > m_nLen )
053     return m_pStr[m_nLen-1];
054   else
055     return m_pStr[n];
056 }
057
058 MyString& MyString::operator=(const MyString &temp)
059 {
060   if(this == &temp)
061     return *this;
062
063   delete [] m_pStr;
064   m_nLen = temp.m_nLen;
065   m_pStr = new char[m_nLen];
066   strcpy(m_pStr, temp.m_pStr);
067   return *this;
068 }
```

```
069
070 MyString::MyString(const MyString & src)
071 {
072   m_nLen=src.m_nLen;
073   m_pStr=new char[m_nLen];
074   strcpy(m_pStr, src.m_pStr);
075 }
076
077 MyString::MyString()
078 {
079   m_nLen=1;
080   m_pStr=new char[m_nLen];
081   strcpy(m_pStr, "");
082 }
083
084 MyString::MyString(const char * const str)
085 {
086   m_nLen = strlen(str)+1;
087   m_pStr = new char[m_nLen];
088   strcpy(m_pStr, str);
089 }
090
091 MyString::~MyString()
092 {
093   delete []m_pStr;
094   m_nLen = 0 ;
095   m_pStr = NULL;
096 }
097
098 ostream & operator<<(ostream & os, MyString & temp)
099 {
100   cout<<temp.m_pStr;
101   return os;
102 }
103
104 void main()
105 {
106   MyString strA("Apple");
107   MyString strB("Banana");
```

```
108
109   if(strA == strB)
110     cout<<"## 두 문자열이 같다.\n";
111   else
112     cout<<"## 두 문자열이 다르다.\n";
113
114   strB="Apple";
115
116   if(strA == strB)
117     cout<<"## 두 문자열이 같다.\n";
118   else
119     cout<<"## 두 문자열이 다르다.\n";
120
121   if(strA == "Apple")
122     cout<<"## 두 문자열이 같다.\n";
123   else
124     cout<<"## 두 문자열이 다르다.\n";
125 }
```

109행, 116행 MyString 객체끼리 == 연산자로 관계를 비교해도 에러가 발생하지 않는다. 23행~39행에서 MyString 객체를 실 매개변수로 하는 == 연산자를 MyString 클래스의 멤버함수로 오버로딩했기 때문이다.

121행 MyString 객체와 문자열 상수를 == 연산자로 관계를 비교해도 에러가 발생하지 않는다. 31행~37행에서 char *를 실 매개변수로 하는 == 연산자를 MyString 클래스의 멤버함수로 오버로딩했기 때문이다.