10장 포인터와 동적 배열

10.1 포인터

* 포인터와 포인터 변수

- 포인터: 어떤 변수의 메모리 주소이다.

- 포인터 변수: 포인터를 저장하는 변수 즉, 어떤 변수의 메모리 주소를 저장하는 변수이다.

* + - 포인터는 비록 2진수이지만, int형, double형 변수에 저장할 수 없어 포인터형 변수를 따로 선언한다.



* + - 각 데이터 형에 따라 포인터 변수를 다르게 선언한다.

typedef

- 기존 테이터 형에 새로운 데이터 형의 이름을 정의하기 위해서 사용한다.

- main 함수와 다른 함수의 외부에 위치한다.

&와 \*

- 주소연산자(&): 변수의 주소를 반환한다.

- 역참조연산자(\*): 포인터가 가리키는 곳(변수)의 값을 리턴 하던지 또는 또 다른 포인터 값을 리턴한다. (포인터의 포인터인 경우)

포인터 연산

- 포인터는 정수나 실수형이 아니라서 산술연산이 안 된다. 단 할당 연산은 가능하다.

- 포인터가 배열인 경우는 덧셈(정수) 연산은 데이터 유형의 크기 단위로 덧셈이 진행된다.

- 주의: 의미가 다르다.

* 예제: 포인터

ex010\_01.cpp

|  |
| --- |
| // 10.1 포인터  #include <iostream>  using namespace std;  int main()  {  // 포인터 변수의 선언  // double \*d1, d2;  //int \*p1, \*p2; // int\* p1, p2;  // int\* p1, \*p2; // 불확실성 제공  // typedef  // typedef int\* IntPtr;  //IntPtr p1, p2; // int\* p1, p2; 같음  // &와 \*  int \*p1, v1;  p1 = &v1;  \*p1 = 3;  cout << v1 << endl;  v1 = 7;  cout << \*p1 << endl;  // 포인터 연산  int \*p2, v2;  p2 = &v2;  //p2 = p1+p2; // 오류  //p2 = p2 + 1; // 불확실한 값을 출력한다. 그러나 배열에서는 의미가 있다.  p2 = p1; // 할당연산자, \*p2 = \*p1; 의미를 구분한다.  cout << \*p2 << endl;  return 0;  } |

* 동적 변수

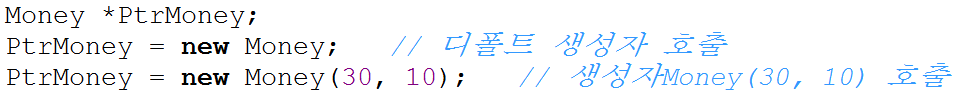
- 동적 변수란 프로그램이 실행되는 동안 생성되는 변수를 의미한다.

new 연산자

- 새로운 이름 없는 변수를 생성하고 그 변수의 포인터를 반환한다.



- 클래스형 동적 변수 생성



* 자유저장공간(free store) 또는 힙(heap)

- 메모리의 일부분이 동적 변수 할당을 위해 사전에 확보된 저장 공간을 말한다.

- 사용 가능한 메모리가 부족하여 더 이상 동적 변수를 생성할 수 없으면 컴파일러에 따라 NULL 포인터를 반환하거나 프로그램을 종료한다.

NULL 포인터

- 포인트 변수에 할당하는 특수한 포인터 값으로서 실제는 정수 0이다. NULL은 아무 것도 가리키지 않는다는 것을 의미한다.

- 그러나 어떤 유형의 포인터 변수에도 사용 가능하다.

- <iostream>에 정의되어 있으며, 문자열의 null문자('\0' - 문자열의 끝)와 구분된다.

delete 연산자

- 동적 변수를 삭제하고 동적 변수가 사용한 메모리를 저유저장공간에게 되돌려준다.



허상 포인터(dangling pointer)

- 삭제된 동적 변수가 가리키고 있는 포인터 변수는 미정상태의 포인터이며 이것을 허상 포인터라고 한다.

- 허상 포인터에 역참조 연산자(\*)을 적용하는 것은 그 결과를 예측할 수 없다.

- 허상 포인터 방지: 삭제된 동적 변수를 가리키는 모든 포인터를 NULL로 설정한다.

동적 변수, 지역(자동) 변수, 정적(전역) 변수

- 동적 변수: 프로그램이 실행되는 동안 생성되는 변수이다.

- 지역(자동) 변수: 함수가 호출될 때 자동으로 할당되고 함수가 종료될 때 자동으로 회수되는 변수로서 우리가 일반적으로 사용하는 변수이다.

- 정적(전역) 변수: 함수나 클래스 외부에 정의된 전역 변수를 정적 변수라고 한다.

* 예제: 동적 변수

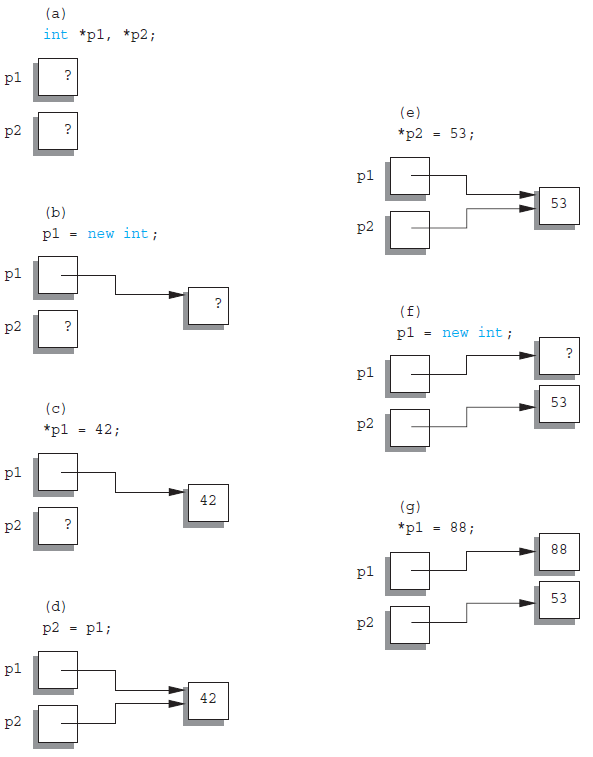
ex010\_02.cpp

|  |
| --- |
| // 10.1 포인터 - 동적 변수  #include <iostream>  using namespace std;  int main()  {  // new 연산자  int \*p1;  p1 = new int; // new double; new char;  \*p1 = 7;  //p1 = new int(7);  cout << \*p1 << endl;  // NULL 포인터  // p1 = NULL;  //cout << p1 << endl; // cout << \*p1; // 오류 why?  // delete 연산자  // 허상 포인터  int \*p2;  p2 = p1;  delete p1; // p2는 허상 포인터  p2 = NULL; // 허상 포인터 방지  cout << p2 << endl; // \*p2: 예측 불가능  return 0;  } |

* 예제: new 연산자

ex010\_03.cpp

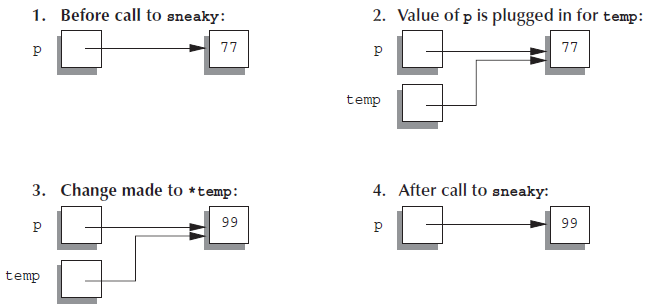
|  |
| --- |
| // 10.1 포인터 - new 연산자 예제  #include <iostream>  using namespace std;  int main()  {  int \*p1, \*p2;    p1 = new int;  \*p1 = 42;  p2 = p1;  cout << "p1=" << \*p1 << " " << "p2=" << \*p2 << endl;  \*p2 = 53;  cout << "p1=" << \*p1 << " " << "p2=" << \*p2 << endl;    p1 = new int;  \*p1 = 88;  cout << "p1=" << \*p1 << " " << "p2=" << \*p2 << endl;  return 0;  } |



* 예제: 포인터를 call-by-value의 인자로 사용

ex010\_04.cpp

|  |
| --- |
| // 10.1 포인터 - 포인터를 call-by-value의 인자로 사용  #include <iostream>  using namespace std;  void sneaky(int \*temp) // void sneaky(int\* temp)  {  \*temp = 99;  cout << "temp=" << \*temp << endl;  }  int main()  {  int \*p;    p = new int;  \*p = 77;  cout << "p=" << \*p << endl;    //int \*temp;  //temp = p;  //\*temp = 99;  //cout << "temp=" << \*temp << endl;  sneaky(p);  cout << "p=" << \*p << endl;  return 0;  } |



10.2 동적 배열

* 배열 변수와 포인터 변수

- 배열 변수(배열명): 배열의 첫 번째 변수를 가리키는 포인트 변수(상수)이다.

- 포인터 변수가 배열의 메모리를 가리키는 한, **대괄호**를 배열과 포인트 변수에 동일하게 사용할 수 있다.

- 사용자는 배열 변수(배열명)의 메모리 주소 값을 변경시킬 수 없다.

- 배열 변수에 대한 산술 연산은 포인터에 대한 산술연산으로서 의미가 있다.

* 예제: 배열 변수와 포인터 변수

ex010\_05.cpp

|  |
| --- |
| // 10.2 동적 배열 - 배열 변수와 포인터변수  #include <iostream>  using namespace std;  int main()  {  // 배열 변수(배열명)  char a[4]={'a', 'b', 'c','d'}; // a는 배열명이며, 배열변수라고 한다.  char \*cp;    // 포인터 상수  cp = a; // a = cp; 오류 why? a는 배열명으로 포인터상수이다.  // 대괄호  cout << cp[0] << cp[1] << endl;    // 포인터의 산술연산  cout << \*cp << \*(cp+1) << endl;  return 0;  } |

* 동적 배열의 생성과 사용

- 동적 배열: 프로그램 실행 중에 배열의 크기를 결정하여 할당한다.

* + - 배열의 크기를 미리 결정하는 경우, 그 크기가 작은 경우 또는 큰 경우에 발생하는 문제점을 해결한다.

- 동적 배열 생성



- 동적 배열 삭제



* + - [] 생략하여도 오류를 출력하지 않는다. 그러나 전체 배열이 삭제되지 않는다.
    - 동적 변수 삭제



* 예제: 숫자 리스트 검색

ex010\_06.cpp

|  |
| --- |
| // 10.2 동적 배열 - 숫자 리스트 검색  #include <iostream>  using namespace std;  int main()  {  // 배열 크기 입력 및 할당  cout << "Size of array= ?";  int asize;  cin >> asize;  int \*pa;  pa = new int[asize];  // 배열 채우기  for(int i=0; i<asize; i++)  cin >> pa[i];  // 배열 검색  cout << "Target= ?";  int tg;  cin >> tg;  int index = -1;  for(int i=0; i<asize; i++)  if (tg == pa[i]) index =i;  // 결과 출력  if (index == -1)  cout << "Your target is not here.\n";  else  cout << "Your target is in " << index << "." << endl;    // 배열 삭제  delete [] pa;  return 0;  } |

* 예제: 숫자 리스트 검색 - 채우기와 검색을 함수로 구현

ex010\_07.cpp

|  |
| --- |
| // 10.2 동적 배열 - 채우기와 검색을 함수로 구현  // ex10\_06.cpp의 예제를 복사하여 사용  #include <iostream>  using namespace std;  //void fillarray(int pa[], int asize)  void fillarray(int \*pa, int asize)  {  for(int i=0; i<asize; i++)  cin >> pa[i];  }  //int search(int pa[], int asize, int tg)  int search(int \*pa, int asize, int tg)  {  for(int i=0; i<asize; i++)  if (tg == pa[i]) return(i);  }  int main()  {  // 배열 크기 입력 및 할당  cout << "Size of array= ?";  int asize;  cin >> asize;  int \*pa;  pa = new int[asize];  // 배열 채우기  fillarray(pa, asize);  // 배열 검색  cout << "Target= ?";  int tg;  cin >> tg;  int index = -1;  index = search(pa, asize, tg);  // 결과 출력  if (index == -1)  cout << "Your target is not here.\n";  else  cout << "Your target is in " << index << "." << endl;    // 배열 삭제  delete [] pa;  return 0;  } |

* 배열을 반환하는 함수

- C++에서 함수가 배열 형을 리턴 할 수 없다. 그러나 함수는 배열의 포인터를 반환할 수 있으므로 배열을 리턴 값으로 얻을 수 있다.





* 예제: 포인터

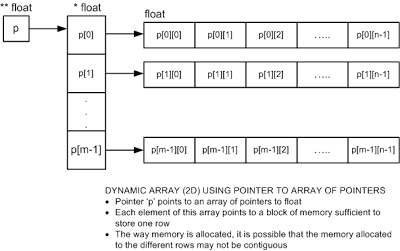
ex010\_08.cpp

|  |
| --- |
| // 10.2 동적 배열 - 배열을 반환하는 함수  #include <iostream>  using namespace std;  void fillarray(int pa[], int asize){  for(int i=0; i< asize; i++)  cin >> pa[i];  }  int\* dbarray(int pa[], int asize)  {  int\* temp = new int[asize];  for(int i=0; i<asize; i++)  temp[i] = 2\*pa[i];  return temp;  }  int main()  {  // 배열 크기 입력 및 할당  cout << "Size of array= ?";  int asize;  cin >> asize;  int \*pa;  pa = new int[asize];    // 배열 채우기  fillarray(pa, asize);  // 배열의 값을 2배하기  int \*pb;  pb = dbarray(pa, asize);  for(int i=0; i< asize; i++)  cout << pb[i] << " ";  cout << endl;  // 배열 삭제  delete [] pa;  delete [] pb;    return 0;  } |

* 다차원 동적 배열

- 동적 다차원 배열은 배열의 배열이다.

|  |
| --- |
| p[m][n]; |

[](http://3.bp.blogspot.com/-Ce4hgph2lJ4/TsLgdbwtg7I/AAAAAAAABms/JsSceuX4IZo/s1600/array_2d_dynamic.png)

- 생성과 삭제 시 번복적인 방법(예 for문)으로 생성하고 삭제한다.

* 예제: 다차원 동적 배열

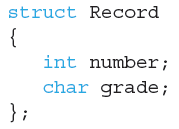
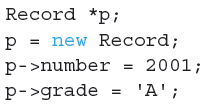
ex010\_09.cpp

|  |
| --- |
| // 10.2 동적 배열 - 다차원 동적 배열: 배열의 배열.  #include<iostream>  using namespace std;  int main()  {  // 배열 크기 입력 및 할당  cout << "Size of matrix(r, c) = ?";  int rw, cl;  cin >> rw >> cl;  // 배열의 생성  int\*\* pm;  pm = new int\* [rw];  for (int i = 0; i < rw; i++)  pm[i] = new int[cl];  // 배열 채우기  for (int i = 0; i < rw; i++)  for (int j = 0; j < cl; j++)  cin >> pm[i][j];  cout << "출력\n";  // 배열 출력  for (int i = 0; i < rw; i++)  {  for (int j = 0; j < cl; j++)  cout << pm[i][j] << " ";  cout << endl;  }  // 배열 반환  for (int i = 0; i < rw; i++)  delete pm[i];  delete [] pm;  return 0;  } |

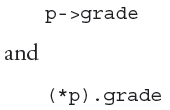
10.3 클래스, 포인터 및 동적 배열

* -> 연산자

- **동적** 구조체나 클래스의 멤버(변수, 함수)를 가리키기 위해 사용하는 \*와 .의 연산자를 하나로 합친 연산자이다.

* + - 동일한 의미



* this 포인터

객체의 저장

- 객체가 생성될 때 멤버 변수와 멤버 함수가 모두 포함된다고 배웠지만 실제적으로는 그렇지 않다.

- 멤버 변수들은 각각 할당되지만 멤버 함수는 **모든 객체가 공유한다.**

- 멤버 함수 호출 시 어떤 객체가 멤버 함수를 호출했는지 구별할 수 있도록 호출한 객체가 함수로 전달하는 포인터이다.

- 이 때 함수는 이 객체를 받기 위한 포인터를 매개변수로 가지게 되는데 이를 this라 한다.

객체1

멤버변수

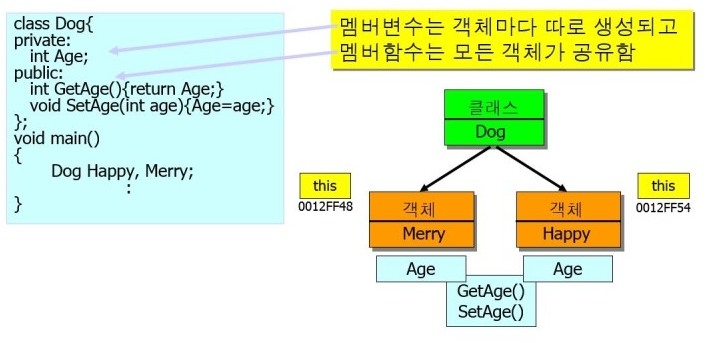
객체2

멤버변수

객체3

멤버변수

멤버 함수



this

- 멤버 함수를 호출한 객체의 주소를 가리키는 상수포인터(시스템 포인터) 이다.

- 멤버 함수는 this라는 상수포인터를 매개변수로 보이지 않게 가지고 있다.

- this는 멤버 함수를 호출한 객체 자신의 주소를 저장하는 포인터이며 이를 자기 참조포인터라 한다.

- 멤버 함수와 멤버 변수를 연결해주는 역할을 한다.

사용도

1. 멤버 변수를 사용하여 멤버 함수나 생성자를 호출하는 경우

2. 멤버 변수에 동적 변수가 있을 경우는 할당 연산자의 오버로딩을 하기 위하여

* + - 만일 할당연산자가 오버로딩 되지 않는다면 기본 할당연산자가 사용되며, 얕은 복사의 문제점을 발생한다.
* 예제: this 포인터와 기본 할당연산자의 문제점

ex010\_10.cpp

|  |
| --- |
| // 10.3 클래스, 포인터 및 동적 배열 -> 연산자, this 포인터  #include<iostream>  using namespace std;  class DArray {  public:  DArray();  DArray(int size);  int getcp() { return cp; }  int getused() { return used; }  void addem(double em) { a[used] = em; used++; }  double getem(int index) { return a[index]; }  // this 포인터  DArray\* getthis() {  return this;  }  // 멤버변수가 멤버함수의 매개변수로 사용된 경우  void dbcp(int cp) { // 현재 cp의 값을 2배로  //cp = 2 \* cp; // 여기서 cp는 매개변수(지역변수), cp가 변하지 않음.  this->cp = 2 \* cp; // (\*this).cp = 2\*cp;  //우선순위: 지역변수 > 멤버변수 > 전역변수  }  // 멤버변수가 멤버함수의 매개변수로 사용되지 않은 경우  //void dbcp(int x) {  // cp = 2 \* x; // 여기서 cp는 멤버변수  //}  private:  double\* a;  int cp; // 용량  int used; // 사용량  };  DArray::DArray() : cp(50), used(0) {  a = new double[cp];  }  DArray::DArray(int size) : cp(size), used(0) {  a = new double[cp];  }  int main()  {  DArray dp1, dp2(10);  // this 포인터  cout << &dp1 << " " << dp1.getthis() << endl;  cout << &dp2 << " " << dp2.getthis() << endl;  // -> 연산자  cout << "dp2의 cp= " << dp2.getcp() << endl;  cout << "dp2의 cp= " << (\*dp2.getthis()).getcp() << endl;  cout << "dp2의 cp= " << dp2.getthis()->getcp() << endl;  // 사용도1: 멤버 변수를 사용하여 멤버 함수나 생성자를 호출하는  dp1.dbcp(dp1.getcp());  cout << "dp1의 cp= " << dp1.getcp() << endl;  // 사용도2: 할당연산자의 오버로딩  // <할당 연산자의 오버로딩이 없는 경우> - 기본 할당연산자 사용  // 문제점: 동적 변수(a)의 주소값을 같이 복사한다.  // 즉, dp1.a과 dp2.a가 같은 곳을 가리킨다.  dp1 = dp2;  dp1.addem(3);  dp2.addem(7);  cout << "cp= " << dp1.getcp() << " " << dp2.getcp() << endl;  cout << "used= " << dp1.getused() << " " << dp2.getused() << endl;  cout << "a[0]= " << dp1.getem(0) << " " << dp2.getem(0) << endl;  return 0;  } |

* 할당연산자의 오버로딩

- 할당연산자는 멤버 함수로 오버로딩 되어야 한다.



|  |
| --- |
| b.operator=(c); |

- 할당연산자의 리턴값이 없어도 된다. 그러나



|  |
| --- |
| a=(b=c);  a=(b.operator=(c)); |

- 할당연산자의 리턴값이 있어야 하며, 왼쪽 항과 같은 테이형이어야 한다. 리턴값을 참조 반환하는 것이 합리적이다.

얕은 복사(shallow copy)와 깊은 복사(deep copy)

할당연산자나 복사연산자를 오버로딩할 때,

- 기본 할당연산자를 이용하여 멤버 변수를 단순 복사하는 것을 ‘얕은 복사’라고 한다.

* + - 포인터나 동적 변수가 없다면 얕은 복사는 잘 작동한다.

- 멤버 변수가 포인터나 동적 변수를 포함하는 경우, 동일한 내용이지만 분리하여 복사하게 하는 것을 ‘깊은 복사’라고 한다.

* + - 각 멤버 변수가 가리키는 것을 복사한다.
* 예제: 할당연산자 오버로딩

ex010\_11.cpp

|  |
| --- |
| // 10.3 클래스, 포인터 및 동적 배열 - 할당 연산자 오버로딩  #include<iostream>  using namespace std;  class DArray {  public:  DArray(); // 용량 = 50  DArray(int size);  int getcp() { return cp; }  int getused() { return used; }  void addem(double em) { a[used] = em; used++;}  double getem(int index) {return a[index];}  // 할당 연산자 오버로딩  DArray& operator = (const DArray& rs);  private:  double\* a;  int cp; // 용량  int used; // 사용량  };  DArray::DArray() : cp(50), used(0) {  a = new double[cp];  }  DArray::DArray(int size) : cp(size), used(0) {  a = new double[cp];  }  // 다른 항인 경우 A = B;  //DArray& DArray::operator = (const DArray& rs) {  // cp = rs.cp;  // used = rs.used;  // delete [] a; // cp와 used가 다를 수 있으므로, 기존 동적배열 a를 반환한다.  // a = new double[cp];  // for (int i = 0; i < used; i++)  // a[i] = rs.a[i];  // return \*this;  //}  // 다른 항과 같은 항인 경우 A = B;, A = A;  DArray& DArray::operator = (const DArray& rs) {  if (this == &rs) {  return \*this;  }  else {  cp = rs.cp;  used = rs.used;  delete[] a; // c기존 동적배열 a를 반환한다.  a = new double[cp];  for (int i = 0; i < used; i++)  a[i] = rs.a[i];  return \*this;  }  }  int main() {  DArray dp1, dp2(10);  // 다른 항인 경우 A = B;  dp1.addem(3); dp1.addem(4);  dp2.addem(7); dp2.addem(8);  cout << "dp1.a= " << dp1.getem(0) << " " << dp1.getem(1) << endl;  cout << "dp2.a= " << dp2.getem(0) << " " << dp2.getem(1) << endl;    //dp1 = dp2; // dp1.operator=(dp2); a=b=c;  //cout << "cp= " << dp1.getcp() << " " << dp2.getcp() << endl;  //cout << "used= " << dp1.getcp() << " " << dp2.getused() << endl;  //cout << "dp1.a= " << dp1.getem(0) << " " << dp1.getem(1) << endl;  //cout << "dp2.a= " << dp2.getem(0) << " " << dp2.getem(1) << endl;  cout << endl;  // 같은 항인 경우  dp1 = dp1;  dp1.addem(9); dp1.addem(10);  cout << "cp= " << dp1.getcp() << endl;  cout << "used= " << dp1.getused() << endl;  cout << "dp1.a= " << dp1.getem(0) << " " << dp1.getem(1) <<" "  << dp1.getem(2) << " " << dp1.getem(3) << endl;  // 얕은 복사: 멤버 변수를 단순 복사(기본 할당자, 기본 복사 연산자)  // 예) cp, used  // 깊은 복사: 멤버 변수가 동적변수일 때, 분리되어 있는 복사를 가지기 위하여  // 각 멤버변수가 가리키는 것을 복사 예) a  return 0;  } |

* 소멸자

- 클래스의 객체가, 스코프를 벗어날 때 자동으로 호출되는 멤버 함수이다.

* + - 동적변수의 경우 자동 반납이 안되기 때문에 소멸자에서 반납한다

- 소멸자는 틸드기호(~)로 시작하며 void도 아니며, 인자도 반환되는 값도 없다. 또한 하나의 클래스에 하나의 소멸자만 가진다.

* 복사 생성자

- 클래스의 생성자의 한 종류로 인자로 주어진 객체와 동일한 객체를 만들어 주는 생성자이다.

* + - 인자는 상수인자로 즉, ‘const 클래스&’ 를 가진다.

- 복사 생성자는 인자로 받아들인 객체의 완벽하고 독립적인 복사를 실행하도록 정의하여야 한다.

- C++이 객체를 복사할 때는 복사 생성자를 자동 호출한다.

- 클래스의 멤버 변수가 포인터나 동적 변수를 포함하면 ‘깊은 복사’를 사용하여 반드시 복사 생성자를 정의하여야 한다.

* + - 그렇지 않으면, 반드시 복사 생성자를 정의할 필요는 없다.

복사 생성자가 호출되는 3가지 경우

1. 객체가 선언되고 괄호 안에서 같은 타입의 다른 객체로 초기화되는 경우

2. 함수가 클래스형의 값을 반환하는 경우

3. 클래스의 객체가 함수의 call-by-value 매개 변수로 넘어가는 경우

* 빅 쓰리(big three)

- 복사 생성자, 할당연산자, 소멸자를 ‘빅 쓰리’라고 부른다.

- 세가지중 하나라도 필요하다면 세가지 모두를 정의해야 한다.

- 멤버 변수로 포인터나 new 연산자를 사용하는 클래스는 빅 쓰리를 반드시 정의해야 한다.

* + - 멤버 변수가 기본 타입(int, double 등)일 때는 문제가 없다.
* 예제: 복사 생성자, 소멸자

ex010\_12.cpp

|  |
| --- |
| /\*10.3 클래스, 포인터 및 동적 배열 - 소멸자, 복사 생성자  복사 생성자가 호출되는 3가지 경우  1. 클래스의 객체가 선언되고 괄호 안에서  가은 타입의 다른 객체로 초기화되는 경우  2. 함수가 클래스형의 값을 반환하는 경우  3. 함수의 call-by-value 매개 변수로 넘어가는 경우  \*/  #include<iostream>  using namespace std;  class DArray {  public:  DArray(); // 용량 = 50  DArray(int size);  // 복사 생성자  DArray(const DArray& tg);  int getcp() { return cp; }  int getused() { return used; }  void addem(double em) { a[used] = em; used++;}  void setem(int index, double em) { a[index] = em; }  double getem(int index) {return a[index];}  // 할당 연산자  DArray& operator = (const DArray& rs);  // 소멸자  ~DArray();  private:  double\* a;  int cp; // 용량  int used; // 사용량  };  DArray::DArray() : cp(50), used(0) {  a = new double[cp];  }  DArray::DArray(int size) : cp(size), used(0) {  a = new double[cp];  }  // 복사 생성자(1)  DArray::DArray(const DArray& tg):cp(tg.cp), used(tg.used)  {  //cp = tg.cp;  //used = tg.used;  a = new double[cp];  for (int i = 0; i < used; i++)  a[i] = tg.a[i];  }  // 복사 생성자 (2) (3)  DArray fnc(DArray a) {  DArray temp;  temp = a;  return temp;  }  // 소멸자  DArray::~DArray() {  delete[] a;  }  // 다른 항과 같은 항인 경우 A = B;, A = A;  DArray& DArray::operator = (const DArray& rs) {  if (this == &rs) {  return \*this;  }  else {  cp = rs.cp;  used = rs.used;  delete[] a; // c기존 동적배열 a를 반환한다.  a = new double[cp];  for (int i = 0; i < used; i++)  a[i] = rs.a[i];  return \*this;  }  }  int main() {  DArray dp1, dp2(10);  // 초기화  dp1.addem(2); dp1.addem(3); dp1.addem(4);  // 복사 생성자 (1): 클래스의 객체가 같은 타입의 다른 객체로 초기화  DArray dp3(dp1);  dp1.setem(0, 10);  cout << "cp= " << dp3.getcp() << endl;  cout << "used= " << dp3.getused() << endl;  cout << "dp3.a= " << dp3.getem(0) << " " << dp3.getem(1) << " "  << dp3.getem(2) << endl;  cout << endl;  // 복사 생성자 (2), (3): 함수의 인자와 리턴값  dp2 = fnc(dp1); // DArray fnc(DArray a)  cout << "cp= " << dp2.getcp() << endl;  cout << "used= " << dp2.getused() << endl;  cout << "dp2.a= " << dp2.getem(0) << " " << dp2.getem(1) << " "  << dp2.getem(2) << endl;  // 빅쓰리: 복사 생성자, 할당 연산자, 소멸자  // 세가지중 하나라도 필요하다면 세가지 모두를 정의해야 한다.  // 멤버 변수가 기본타입(int, double 등)은 문제가 없다.  // 그러나, 포인터나 new연산자를 사용하는 클래스는 빅쓰리를  // 반드시 정의해야 한다.  return 0;  } |