14장 상속

14.1 상속의 기초

- 상속은 기반 클래스(base class, 부모 클래스, parent class)라 불리는 클래스로부터 파생 클래스(derived class, 자식 클래스, child class)라 불리는 새로운 클래스를 생성하는 과정이다.

* 파생 클래스

- 파생 클래스 정의: 콜론(:)과 예약어 public 추가



- 파생 클래스는 기반 클래스가 가진 모든 멤버 변수와 멤버 함수를 자동적으로 가진다.

* + - 또한 추가적으로 새로운 멤버 함수와 멤버 변수를 가질 수 있다.
    - 생성자(소멸자, 복사 생성자, 할당 연산자)와 같은 일부 특수한 멤버 함수는 상속되지 않는다.

- 상속된 멤버 함수의 정의를 바꾸려면 파생 클래스에서 재정의해야 한다.

- 상속을 통해 기반 클래스의 코드를 재사용할 수 있다.

접근 제한자

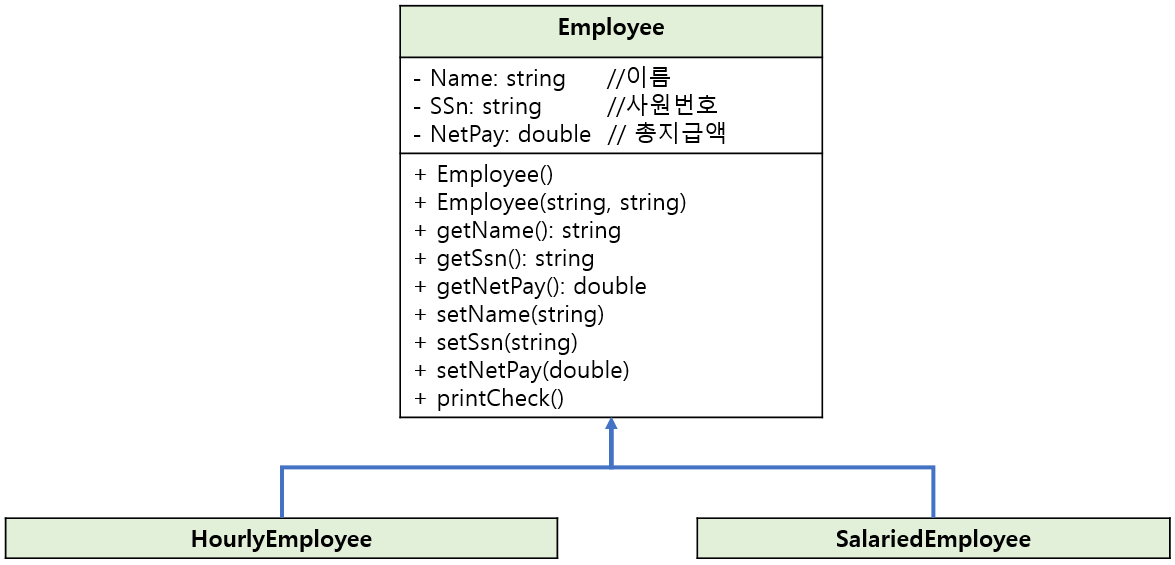


- public: 모든 다른(외부) 클래스에서 접근 가능

- protected: 다른(외부) 클래스는 차단하고, 상속되는 클래스만 접근 가능

- private: 다른(외부) 클래스는 차단하고, 클래스 내부에서만 접근 가능

예제: 상속 - 기반 클래스

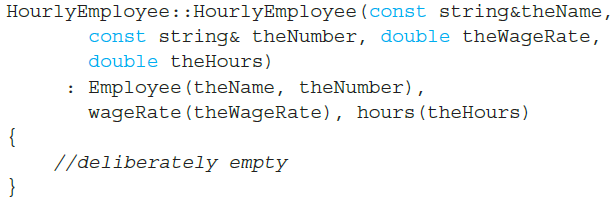


ex14\_01.cpp

|  |
| --- |
| // 14.1 상속의 기초  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setName(const string& newName);  void setSsn(const string& newSsn);  void setNetPay(double newNetPay);  void printCheck() const;  private:  string Name;  string Ssn;  double NetPay;  };  // main 함수  int main() {    Employee e1, e2("Kim", "0103");  e1.setName("Park");  e1.setSsn("0209");  cout << "<e1의 정보 출력>\n";  cout << e1.getName() << endl;  cout << e1.getSsn() << endl;  cout << e1.getNetPay() << endl;  e1.printCheck();  cout << endl;  cout << "<e2의 정보 출력>\n";  cout << e2.getName() << endl;  cout << e2.getSsn() << endl;  cout << e2.getNetPay() << endl;  e2.printCheck();  cout << endl;  return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setName(const string& newName)  {  Name = newName;  }  void Employee::setSsn(const string& newSsn)  {  Ssn = newSsn;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  void Employee::printCheck() const  {  cout << "Your check is not determined yet.";  } |

* 파생 클래스의 생성자

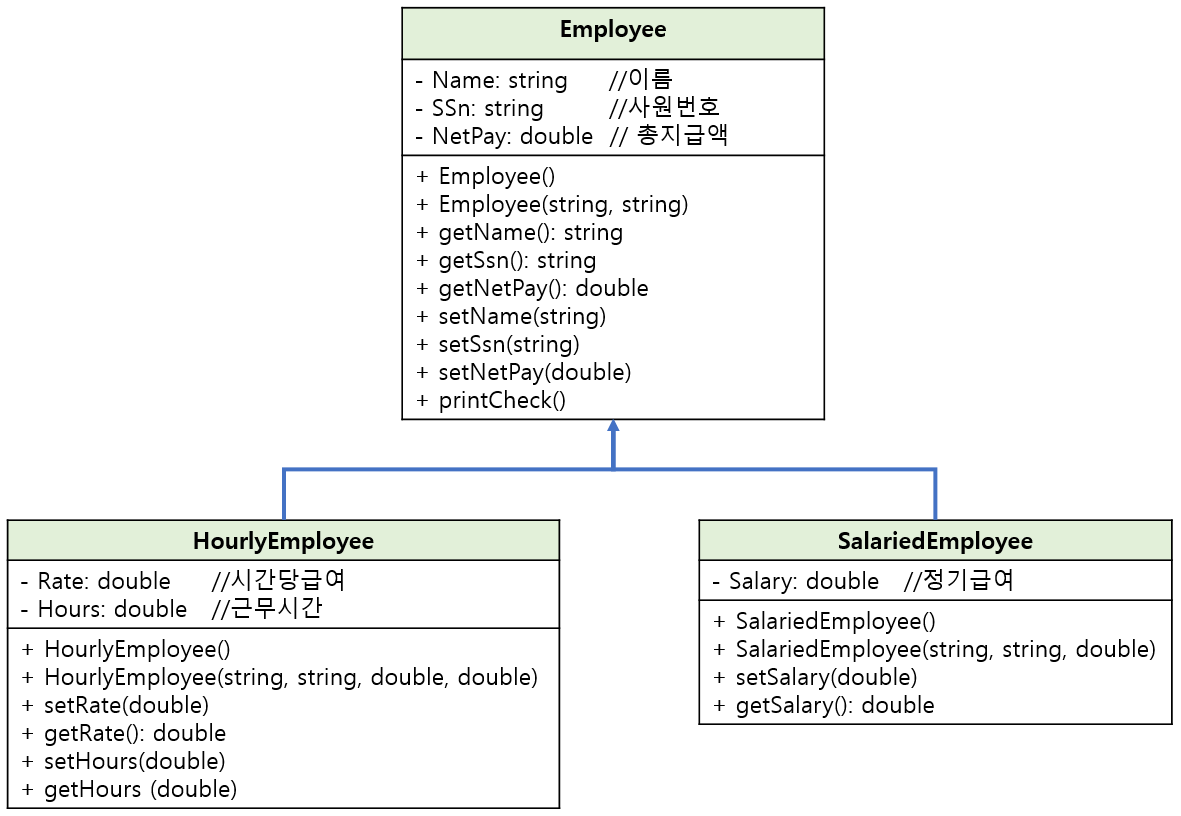
- 기반 클래스의 생성자는 파생 클래스에 상속되지 않는다. 대신 파생 클래스의 생성자에서 기반 클래스의 생성자를 호출할 수 있다.



* + - 기반 클래스의 생성자를 호출하는 문법은 멤버 변수의 값을 설정하는 초기화 섹션과 유사하다.
    - 기반 클래스의 생성자가 호출되지 않는다면 디폴트 생성자가 자동으로 호출된다.
    - 호출 순서는 상위 기반 클래스부터 호출된다.

- 기반 클래스가 사용될 수 있는 곳에는 파생 클래스도 사용할 수 있다. 즉, 파생 클래스의 객체는 하나 이상의 타입을 가진다.

예제: 파생 클래스 생성자



ex14\_02.cpp

|  |
| --- |
| /\* 14.1 상속  Employee 클래스를 상속받는 HourlyEmployee 클래스를 구현하는  추가되는 멤버 변수: double Hours, double Rate  멤버 함수: double getHours() const;  double getRate() const;  void setHours(double hoursWorked);  void setRate(double newRate);  파생 클래스의 생성자 구현  \*/  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setName(const string& newName);  void setSsn(const string& newSsn);  void setNetPay(double newNetPay);  void printCheck() const;  private:  string Name;  string Ssn;  double NetPay;  };  // HourlyEmployee 클래스 인터페이스  /\* 파생 클래스는 추가되는 멤버 변수와 함수만 정의한다. \*/  class HourlyEmployee : public Employee {  /\* 기반 클래스의 생성자는 파생 클래스에 상속되지 않는다. 대신 파생 클래스의 생성자에서  생성자에서 기반 클래스의 생성자를 호출할 수 있다. \*/  public:  HourlyEmployee();  HourlyEmployee(const string& theName, const string& theSsn,  double theRate, double theHours);  double getHours() const;  double getRate() const;  void setHours(double hoursWorked);  void setRate(double newRate);  // 기반 클래스의 멤버 변수는 자동으로 상속받는다.  private:  double Hours;  double Rate;  };  // main 함수  int main() {    HourlyEmployee h1, h2("Hong", "0329", 10, 30);  h1.setName("Kang");  h1.setSsn("0308");  h1.setRate(20);  h1.setHours(15);  cout << "<h1의 정보 출력>\n";  cout << h1.getName() << endl;  cout << h1.getSsn() << endl;  cout << h1.getHours() << endl;  cout << h1.getRate() << endl;  cout << "<h2의 정보 출력>\n";  cout << h2.getName() << endl;  cout << h2.getSsn() << endl;  cout << h2.getHours() << endl;  cout << h2.getRate() << endl;  cout << endl;  // 기반 클래스가 사용될 수 있는 곳에는 파생 클래스도 사용할 수 있다.  // 즉, 파생 클래스의 객체는 하나 이상의 타입을 가진다.  Employee e1, e2("Kim", "0103");  e1 = h2;  //cout << "<e2의 정보 출력>\n";  cout << e1.getName() << endl;  cout << e1.getSsn() << endl;  return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setName(const string& newName)  {  Name = newName;  }  void Employee::setSsn(const string& newSsn)  {  Ssn = newSsn;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  void Employee::printCheck() const  {  cout << "Your check is not determined yet.";  }  // HourlyEmployee 클래스 구현  /\* 기반 클래스의 생성자를 호출하는 문법은 멤버 변수의 값을 설정하는 문법과  유사하다. 만약 기반 클래스의 생성자를 호출하는 부분이 없으면, 기반 클래스의  디폴트 생성자 Employee() 을 자동 호출한다.  Employee() 없다면, 오류  호출 순서: 파생 순서와 같다.\*/  // 생성자  HourlyEmployee::HourlyEmployee() : Employee(), Rate(0), Hours(0)  {  }  HourlyEmployee::HourlyEmployee(const string& theName, const string& theSsn,  double theRate, double theHours) : Employee(theName, theSsn), Rate(theRate), Hours(theHours)  {  }  void HourlyEmployee::setRate(double newRate)  {  Rate = newRate;  }  double HourlyEmployee::getRate() const  {  return Rate;  }  void HourlyEmployee::setHours(double hoursWorked)  {  Hours = hoursWorked;  }  double HourlyEmployee::getHours() const  {  return Hours;  } |

* 상속된 함수의 재정의

- 상속된 멤버 함수에 대한 새로운 구현을 원한다면, 파생 클래스에 선언하고 재정의하면 된다.

* + - 비록 파생 클래스라고 할지라도 기반 클래스의 private 멤버 변수와 함수는 직접 접근이 불가능하다.
* protected 제한자

- 기반 클래스의 멤버 변수나 함수가 protected 제한자로 선언되었다면, 파생 클래스에서는 직접 접근이 가능하다. 그러나 다른 클래스에서는 private 효과를 가진다.

- protected 멤버들은 파생 클래스에서 protected로 표기된 것처럼 또다시 상속된다.

* 오버로딩 대 재정의

- 기반 클래스에 있는 멤버 함수를 파생 클래스에서 같은 이름의 멤버 함수를 추가함으로써 함수 오버로딩을 구현할 수 있다.

* + - 기반 클래스의 함수와 같은 이름이지만, 매개 변수의 수나 타입이 다른 함수를 파생 클래스에 추가하는 것은 재정의가 아니라 함수 오버로딩이다.

 기반 클래스

 파생 클래스

* 파생 클래스에서 기반 클래스 함수에 대한 접근

- 파생 클래스의 객체에서 기반 클래스에서 정의된 함수를 호출하고자 한다면, 기반 클래스 이름으로 영역 지정 연산자(::)를 사용한다.





예제: 함수 재정의, protected 제한자, 함수 오버로딩

ex14\_03.cpp

|  |
| --- |
| /\* 14.1 상속  상속된 함수의 재정의  protected 제한자  파생 클래스의 함수 overloading  파생 클래스에서 기반 클래스 함수에 대한 접근  \*/  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setName(const string& newName);  void setSsn(const string& newSsn);  void setNetPay(double newNetPay);  void printCheck() const; // 실제 정의는 파생 클래스에 있다.  //private:  protected: // 상속되는 클래스는 직접 접근이 가능  string Name;  string Ssn;  double NetPay;  };  // HourlyEmployee 클래스 인터페이스  /\* 함수의 재정의: 상속된 멤버 함수에 대한 새로운 구현을 원한다면,  파생 클래스에 선언하고 재정의하면 된다. \*/  class HourlyEmployee : public Employee {  public:  HourlyEmployee();  HourlyEmployee(const string& theName, const string& theSsn,  double theRate, double theHours);  double getHours() const;  double getRate() const;  void setHours(double hoursWorked);  void setRate(double newRate);  // 함수 재정의  void printCheck() ;  // 함수 overloading  // void setName(const string& first, const string& last)  // first와 last를 위한 이름 변수가 필요함. 추가 코드 필요  private:  double Hours;  double Rate;  };  // main 함수  int main() {    HourlyEmployee h1, h2("Hong", "0329", 10, 20);  h1.setName("Kang");  h1.setSsn("0308");  h1.setRate(20);  h1.setHours(15);  cout << "<h1의 정보 출력>\n";  cout << h1.getName() << endl;  cout << h1.getSsn() << endl;  cout << h1.getHours() << endl;  cout << h1.getRate() << endl;  h1.printCheck();  cout << endl;  cout << "<h2의 정보 출력>\n";  cout << h2.getName() << endl;  cout << h2.getSsn() << endl;  cout << h2.getHours() << endl;  cout << h2.getRate() << endl;  h2.printCheck();  cout << endl;  // 파생 클래스에서 기반 클래스 함수에 대한 접근  h2.Employee::printCheck();  cout << endl;    return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setName(const string& newName)  {  Name = newName;  }  void Employee::setSsn(const string& newSsn)  {  Ssn = newSsn;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  void Employee::printCheck() const  {  cout << "Your check is not determined yet.";  }  // HourlyEmployee 클래스 구현  HourlyEmployee::HourlyEmployee() : Employee(), Rate(0), Hours(0)  {  }  HourlyEmployee::HourlyEmployee(const string& theName, const string& theSsn,  double theRate, double theHours) : Employee(theName, theSsn), Rate(theRate), Hours(theHours)  {  }  void HourlyEmployee::setRate(double newRate)  {  Rate = newRate;  }  double HourlyEmployee::getRate() const  {  return Rate;  }  void HourlyEmployee::setHours(double hoursWorked)  {  Hours = hoursWorked;  }  double HourlyEmployee::getHours() const  {  return Hours;  }  // 함수 재정의  //// 비록 파생 클래스라고 할지라도 기반 클래스의 private 멤버 변수의 접근을 못한다.  //  //void HourlyEmployee::printCheck()  //{  // setNetPay(Hours \* Rate);  // cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  // cout << "Name: " << getName() << endl;  // cout << "Number: " << getSsn() << endl;  // cout << "Your are an hourly employee." << endl;  // cout << "You worked for " << Hours << " with the rate " << Rate << endl;  // cout << "NetPay: $" << getNetPay() << endl;  // cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  //}  // protected 선언: private 멤버에 직접 접근이 가능  void HourlyEmployee::printCheck()  {  NetPay = (Hours \* Rate);  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  cout << "Name: " << Name << endl;  cout << "Number: " << Ssn << endl;  cout << "Your are an hourly employee." << endl;  cout << "You worked for " << Hours << " with the rate " << Rate << endl;  cout << "NetPay: $" << NetPay << endl;  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  } |

* SalariedEmployee 클래스

- Employee 클래스를 상속 받는 SalariedEmployee 클래스를 구현하라.

- 추가 되는 멤버 변수와 함수는 다음과 같다.

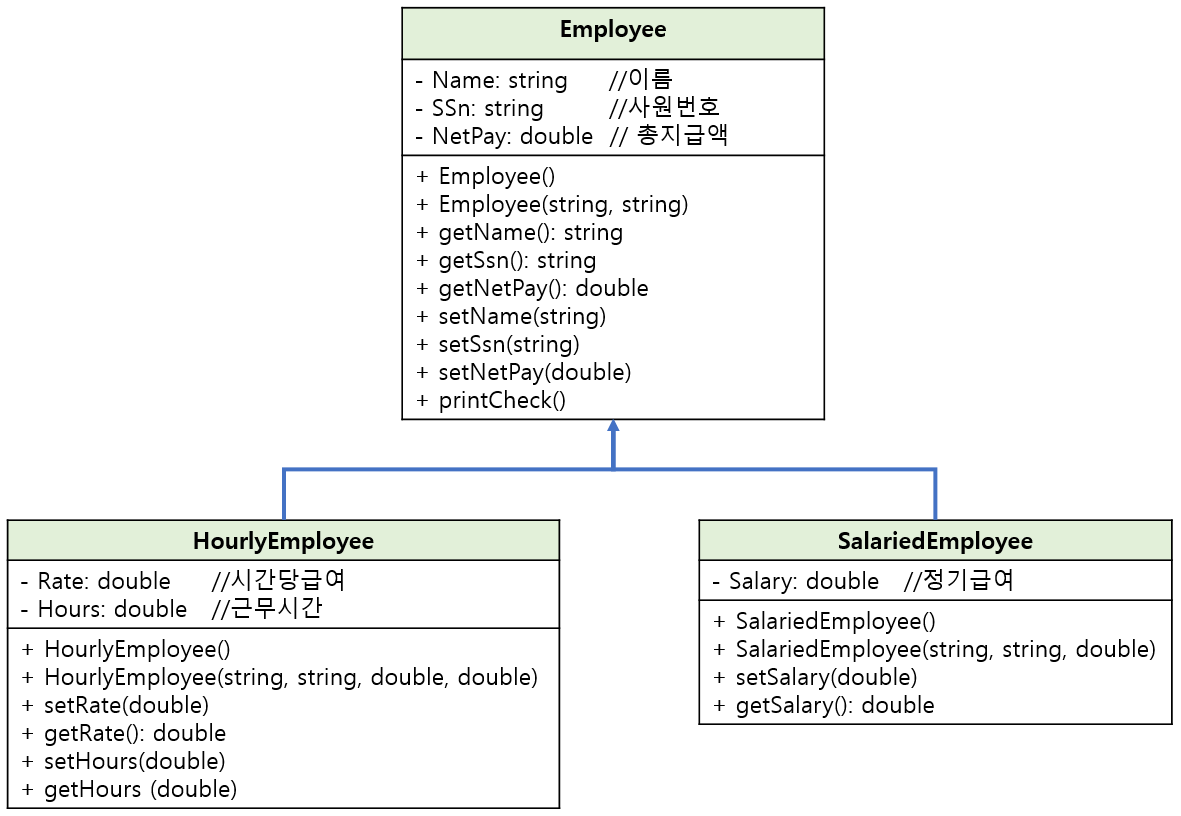
* + - 변수: double Salary;
    - 멤버 함수: void setSalary(double newSalary);

double getSalary() const;

void printCheck() const; // 함수 재정의(기본 클래스의 멤버 변수는 private)

2 개의 SalariedEmployee 생성자

예제: SalariedEmployee 클래스



ex14\_04.cpp

|  |
| --- |
| /\* 14.1 상속  Employee 클래스를 상속받는 SalariedEmployee 클래스를 구현하라.  추가되는 멤버 변수: double Salary;  멤버 함수: void setSalary(double newSalary);  double getSalary() const;  void printCheck() const;  2개의 SalariedEmployee 생성자  \*/  #include<iostream>  #include<string>  using namespace std;  // Employee 클래스 인터페이스  class Employee  {  public:  Employee();  Employee(const string& theName, const string& theSsn);  string getName() const;  string getSsn() const;  double getNetPay() const;  void setName(const string& newName);  void setSsn(const string& newSsn);  void setNetPay(double newNetPay);  void printCheck() const;  private:  string Name;  string Ssn;  double NetPay;  };  // SalariedEmployee 클래스 인터페이스  class SalariedEmployee :public Employee  {  public:  SalariedEmployee();  SalariedEmployee(const string& theName, const string& theSsn, double theSalary);  void setSalary(double newSalary);  double getSalary() const;  void printCheck();  private:  double Salary;  };  // main 함수  int main() {    SalariedEmployee s1, s2("Lee", "0508", 500);  s1.setName("Yang");  s1.setSsn("0606");  s1.setSalary(700);  cout << "<s1의 정보 출력>\n";  cout << s1.getName() << endl;  cout << s1.getSsn() << endl;  cout << s1.getSalary() << endl;  s1.printCheck();  cout << endl;  cout << "<s2의 정보 출력>\n";  cout << s2.getName() << endl;  cout << s2.getSsn() << endl;  cout << s2.getSalary() << endl;  s2.printCheck();  cout << endl;    return 0;  }  // Employee 클래스 구현  Employee::Employee() : Name("No name"), Ssn("No number"), NetPay(0){  }  Employee::Employee(const string& theName, const string& theNumber):  Name(theName), Ssn(theNumber), NetPay(0)  {}  string Employee::getName() const  {  return Name;  }  string Employee::getSsn() const  {  return Ssn;  }  double Employee::getNetPay() const  {  return NetPay;  }  void Employee::setName(const string& newName)  {  Name = newName;  }  void Employee::setSsn(const string& newSsn)  {  Ssn = newSsn;  }  void Employee::setNetPay(double newNetPay)  {  NetPay = newNetPay;  }  void Employee::printCheck() const  {  cout << "Your check is not determined yet.";  }  // SalariedEmployee 클래스 구현  SalariedEmployee::SalariedEmployee() :Employee(), Salary(0)  {  }  SalariedEmployee::SalariedEmployee(const string& theName,  const string& theSsn, double theSalary):Employee(theName, theSsn), Salary(theSalary)  {  }  void SalariedEmployee::setSalary(double newSalary)  {  Salary = newSalary;  }  double SalariedEmployee::getSalary() const  {  return Salary;  }  void SalariedEmployee::printCheck()  {  setNetPay(Salary);  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  cout << "Name: " << getName() << endl;  cout << "Number: " << getSsn() << endl;  cout << "You are a salaried employee." << endl;  cout << "Your weekly salary is " << getNetPay() << endl;  cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";  } |

14.2 상속을 이용한 프로그래밍

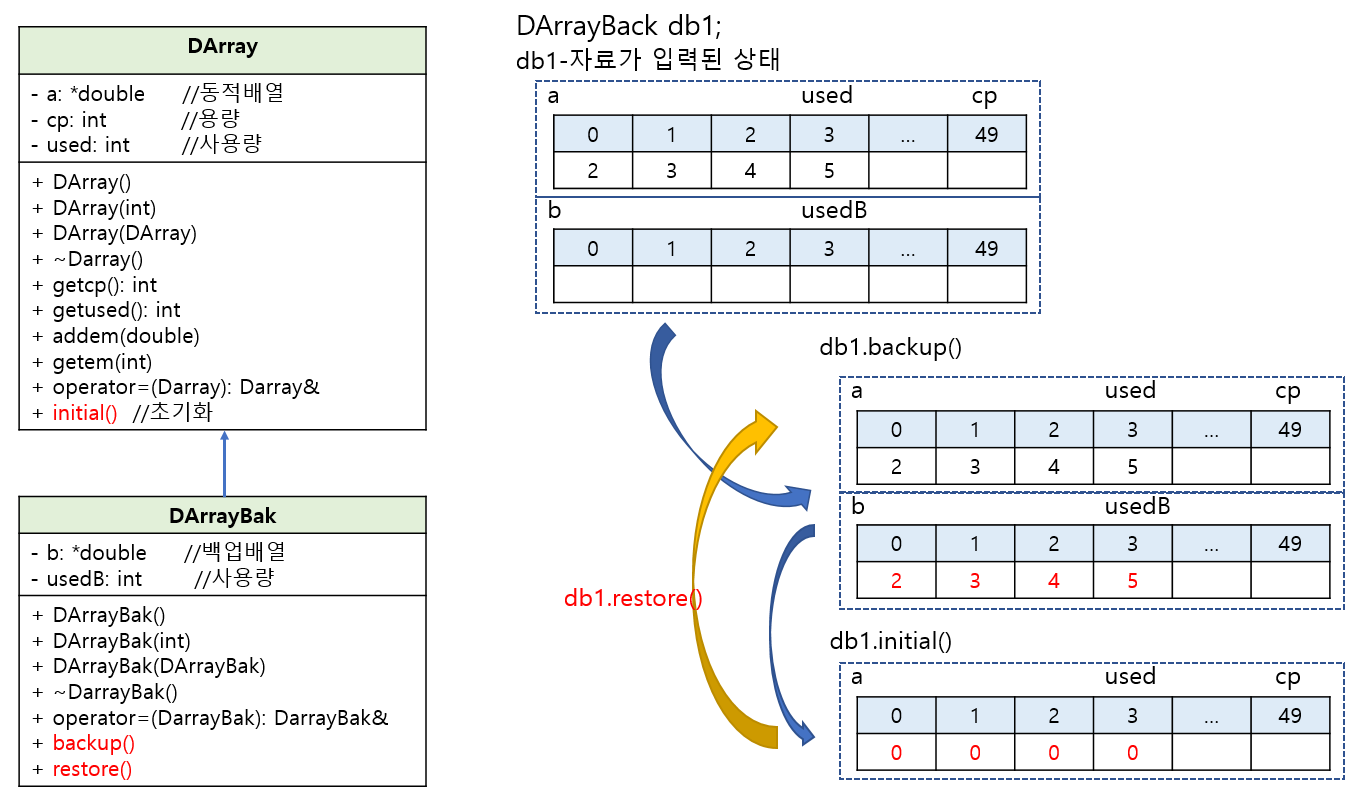
* 10장의 부분적으로 재워진 배열 상속

- 파생 클래스는 기반 클래스의 내용을 백업 또는 복구할 수 있다.

- 동적 변수를 포함한 클래스의 상속의 경우

* + - 파생 클래스의 복사 생성자, 할당 연산자, 소멸자는 기반 클래스의 복사 생성자, 할당 연산자, 소멸자를 호출해야 하며 추가적인 작업을 해야 한다.

예제: 동적 변수를 포함한 클래스의 상속



ex14\_05.cpp

|  |
| --- |
| /\*14.2 상속을 이용한 프로그래밍  10장의 부분적으로 채워진 배열 상속  동적 변수를 포함한 클래스의 상속  -파생 클래스의 복사 생성자: 기반 클래스의 복사 생성자 호출  -파생 클래스의 할당 연산자: 기반 클래스의 할당 연산자 호출  -파생 클래스의 소멸자: 기반 클래스의 소멸자 자동 호출  \*/  #include<iostream>  using namespace std;  //--- 기본 클래스 DArray 인터페이스  class DArray {  public:  DArray(); // 용량=50  DArray(int size);  DArray(const DArray& tg); //복사 생성자  int getcp() { return cp; }  int getused() { return used; }  void addem(double em) { a[used] = em; used++; }  double getem(int index) { return a[index]; }  void initial(); // 배열 초기화  DArray& operator = (const DArray& rs); // 할당 연산자  ~DArray();  protected: //private 대신  double\* a;  int cp; //용량  int used; //사용량  };  //--- 파생클래스 DArrayBack 인터페이스  class DArrayBack:public DArray// 상속  {  // 상속  public:  // 생성자  DArrayBack(); //용량 = 50  DArrayBack(int cpb);  // 복사 생성자  DArrayBack(const DArrayBack& object);  void backup(); // 백업  void restore(); // 복구  // 할당 연산자  DArrayBack& operator = (const DArrayBack& rsb);  // 소멸자  ~DArrayBack();  private:  double\* b; // 백업 배열  int usedB; // 백업 배열의 사용량  };  //---파생클래스 DArrayBack 구현  DArrayBack::DArrayBack() : DArray(), usedB(0) {  b = new double[cp];  }  DArrayBack::DArrayBack(int cpb) : DArray(cpb), usedB() {  b = new double[cp];  }  // 복사 생성자: 기반 클래스의 복사 생성자 호출  DArrayBack::DArrayBack(const DArrayBack& object) : DArray(object), usedB(0)  {  b = new double[cp];  usedB = object.usedB;  for (int i = 0; i < usedB; i++)  b[i] = object.b[i];  }  void DArrayBack::backup() {  usedB = used;  for (int i = 0; i < usedB; i++)  b[i] = a[i];  }  void DArrayBack::restore() {  used = usedB;  for (int i = 0; i < used; i++)  a[i] = b[i];  }  // 할당 연산자 오버로딩  DArrayBack& DArrayBack::operator = (const DArrayBack& rsb)  {  DArray::operator=(rsb); // 기반 클래스의 할당연산자 호출  if (cp != rsb.cp)  {  delete[] b;  b = new double[rsb.cp];  }  usedB = rsb.usedB;  for (int i = 0; i < usedB; i++)  b[i] = rsb.b[i];  return \*this;  }  // 파생 클래스의 소멸자는 기반 클래스의 소멸자를 자동 호출한다.  // 소멸자 호출 순서는 생성자 호출 순서의 역순  DArrayBack::~DArrayBack()  {  delete [] b;  }  int main() {  DArrayBack db1, db2(10);  // 백업, 복구  db1.addem(2); db1.addem(3); db1.addem(4); db1.addem(5);  db1.backup(); //백업  db1.initial(); //초기화  cout << "db1.a= " << db1.getem(0) << " " << db1.getem(1) << " " << db1.getem(2) << " " << db1.getem(3) << endl;  db1.restore(); // 복구  cout << "db1.a= " << db1.getem(0) << " " << db1.getem(1) << " " << db1.getem(2) << " " << db1.getem(3) << endl;  // 복사 생성자  DArrayBack db3(db1);  cout << "db3.a= " << db3.getem(0) << " " << db3.getem(1) << " " << db3.getem(2) << " " << db1.getem(3) <<endl;  // 할당 연산자(=)  db2 = db3;  cout << "db2.cp= " << db2.getcp() << " db2.used= " << db2.getused() << endl;  cout << "db2.a= " << db2.getem(0) << " " << db2.getem(1) << " " << db2.getem(2) << " " << db1.getem(3) << endl;  return 0;  }  //---기본 클래스 DArray 구현  DArray::DArray() : cp(50), used(0) {  a = new double[cp];  }  DArray::DArray(int size) : cp(size), used(0) {  a = new double[cp];  }  // 복사 생성자  DArray::DArray(const DArray& tg) : cp(tg.cp), used(tg.used) {  a = new double[cp];  for (int i=0; i < used; i++)  a[i] = tg.a[i];  }  // 할당연산자(=) 오버로딩  DArray& DArray::operator = (const DArray& rs) {  if (this == &rs) {  return \*this;  }  else {  cp = rs.cp;  used = rs.used;  delete[] a; // 기존 동적 배열 a를 반환한다.  a = new double[cp];  for (int i = 0 ;i < used; i++)  a[i] = rs.a[i];  return \*this;  }  }  void DArray::initial() {  for (int i = 0; i < used; i++)  a[i] = 0.0;  used = 0;  }  //소멸자  DArray::~DArray() {  delete [] a;  } |

예제: protected 제한자를 private으로 변경

* + - 기반 클래스의 멤버 변수를 직접 접근 할 수 없기에 기반 클래스의 멤버 함수를 이용해야 한다

ex14\_06.cpp

|  |
| --- |
| /\*14.2 상속을 이용한 프로그래밍: ex14\_05.cpp의 결과를 수정하여 실습  \*/  #include<iostream>  using namespace std;  //--- 기본 클래스 DArray 인터페이스  class DArray {  public:  DArray(); // 용량=50  DArray(int size);  DArray(const DArray& tg); //복사 생성자  int getcp() const { return cp; }  int getused() { return used; }  void addem(double em) { a[used] = em; used++; }  double getem(int index) { return a[index]; }  void initial(); // 배열 초기화  DArray& operator = (const DArray& rs); // 할당 연산자  ~DArray();  private:  double\* a;  int cp; //용량  int used; //사용량  };  //--- 파생클래스 DArrayBack 인터페이스  class DArrayBack:public DArray// 상속  {  // 상속  public:  // 생성자  DArrayBack(); //용량 = 50  DArrayBack(int cpb);  // 복사 생성자  DArrayBack(const DArrayBack& object);  void backup(); // 백업  void restore(); // 복구  // 할당 연산자  DArrayBack& operator = (const DArrayBack& rsb);  // 소멸자  ~DArrayBack();  private:  double\* b; // 백업 배열  int usedB; // 백업 배열의 사용량  };  //---파생클래스 DArrayBack 구현  DArrayBack::DArrayBack() : DArray(), usedB(0) {  b = new double[getcp()];  }  DArrayBack::DArrayBack(int cpb) : DArray(cpb), usedB() {  b = new double[getcp()];  }  // 복사 생성자: 기반 클래스의 복사 생성자 호출  DArrayBack::DArrayBack(const DArrayBack& object) : DArray(object), usedB(0)  {  b = new double[getcp()];  usedB = object.usedB;  for (int i = 0; i < usedB; i++)  b[i] = object.b[i];  }  void DArrayBack::backup() {  usedB = getused();  for (int i = 0; i < usedB; i++)  b[i] = getem(i);  }  void DArrayBack::restore() {  // setused(usedB);  for (int i = 0; i < usedB; i++)  addem(b[i]);  }  // 할당 연산자 오버로딩  DArrayBack& DArrayBack::operator = (const DArrayBack& rsb)  {  DArray::operator=(rsb); // 기반 클래스의 할당연산자 호출  if (getcp() != rsb.getcp()) // getcp()에 const 추가  {  delete[] b;  b = new double[rsb.getcp()];  }  usedB = rsb.usedB;  for (int i = 0; i < usedB; i++)  b[i] = rsb.b[i];  return \*this;  }  // 파생 클래스의 소멸자는 기반 클래스의 소멸자를 자동 호출한다.  // 소멸자 호출 순서는 생성자 호출 순서의 역순  DArrayBack::~DArrayBack()  {  delete [] b;  }  int main() {  DArrayBack db1, db2(10);  // 백업, 복구  db1.addem(2); db1.addem(3); db1.addem(4); db1.addem(5);  db1.backup(); //백업  db1.initial(); //초기화  cout << "db1.a= " << db1.getem(0) << " " << db1.getem(1) << " " << db1.getem(2) << " " << db1.getem(3) << endl;  db1.restore(); // 복구  cout << "db1.a= " << db1.getem(0) << " " << db1.getem(1) << " " << db1.getem(2) << " " << db1.getem(3) << endl;  // 복사 생성자  DArrayBack db3(db1);  cout << "db3.a= " << db3.getem(0) << " " << db3.getem(1) << " " << db3.getem(2) << " " << db1.getem(3) <<endl;  // 할당 연산자(=)  db2 = db3;  cout << "db2.cp= " << db2.getcp() << " db2.used= " << db2.getused() << endl;  cout << "db2.a= " << db2.getem(0) << " " << db2.getem(1) << " " << db2.getem(2) << " " << db1.getem(3) << endl;  return 0;  }  //---기본 클래스 DArray 구현  DArray::DArray() : cp(50), used(0) {  a = new double[cp];  }  DArray::DArray(int size) : cp(size), used(0) {  a = new double[cp];  }  // 복사 생성자  DArray::DArray(const DArray& tg) : cp(tg.cp), used(tg.used) {  a = new double[cp];  for (int i=0; i < used; i++)  a[i] = tg.a[i];  }  // 할당연산자(=) 오버로딩  DArray& DArray::operator = (const DArray& rs) {  if (this == &rs) {  return \*this;  }  else {  cp = rs.cp;  used = rs.used;  delete[] a; // 기존 동적 배열 a를 반환한다.  a = new double[cp];  for (int i = 0 ;i < used; i++)  a[i] = rs.a[i];  return \*this;  }  }  void DArray::initial() {  for (int i = 0; i < used; i++)  a[i] = 0.0;  used = 0;  }  //소멸자  DArray::~DArray() {  delete [] a;  } |

* private 멤버의 접근

- 어떤 클래스의 정의(특히 복사 생성자, 할당 연산자)(클래스 범위)에서, 그 클래스와 같은 형의 클래스에 대한 private 멤버 변수의 직접 접근이 가능하다.

* 기반 및 상속 클래스의 관계

is-a: 단순한 클래스에서 복잡한 클래스로 확장

예) Employee --> HourlyEmployee

has-a: 한 클래스가 다른 클래스를 포함

예) Employee 클래스가 Data 클래스를 포함

* 다중 상속

- 하나 이상의 기반 클래스를 가지는 경우를 의미하며, 콤마로 구분한다.

- 충분한 지식이 없으면, 피한다.