8장 연산자 오버로딩, 프렌드 함수 그리고 참조

8.1 기본 연산자 오버로딩

‘+’나 ‘==’ 같은 연산자를 오버로딩 하게 하여 클래스의 객체에 적용되게 한다.

* + - 일반(전역) 함수 형태의 오버로딩
    - 멤버 함수로서의 오버로딩
* 일반(전역) 함수 형태의 오버로딩

- 연산자 기호(+, -, ++ 등)가 함수 이름으로 사용될 수 있고, 일반 연산자보다 우선권이 있다.

* + - 예약어 ‘operator’를 사용한다.

- 매개변수(피연자)가 최소 하나의 클래스 형이면 다른 매개변수(피연자)는 어떤 형도 될 수 있다.

- 일반 함수의 기능을 다 가지고 있다.

* + - call-by-value나 call-by-reference도 될 수 있으나, call-by-reference가 더 효율적이다.
    - 리턴형은 클래스를 포함한 어떤 형도 될 수 있다.

- 2항 연산자의 교환 법칙이 성립한다.

const(상수) 값에 의한 리턴

- 리턴되는 객체를 변경할 수 없게 한다.

- int, double, char와 같은 기본형에 적용하는 것은 무의미하며, 멤버 변수를 포함한 클래스는 변경될 수 있기 때문에 const는 리턴 값의 변경을 방지한다.

예제: 전역 함수에 의한 연산자 오버로딩

ex08\_01.cpp

|  |
| --- |
| #include <iostream> // 연산자 오버로딩: 일반(전역) 함수형태  using namespace std;  class Money{  public:  Money(int pdollars=0, int pcents=0);  void get\_Money() const;  void set\_Money();  Money set\_return(); // set and return  int get\_dollars() const;  int get\_cents() const;  private:  int dollars;  int cents;  };  // 일반(전역) 함수 연산자 오버로딩  //Money operator + (Money a, Money b) // 1. Call by value 형  //Money operator + (Money& a, Money& b) // 2. 효율성을 위하여 참조연산자 형  //Money operator + (const Money& a, const Money& b) // 3. 멤버변수 변경 방지  const Money operator + (const Money& a, const Money& b) // 4. const에 의한 리턴값 변경 방지  {  int allcents = a.get\_cents() + b.get\_cents();  int finalcents = allcents % 100;  int finaldollars = a.get\_dollars() + b.get\_dollars() + (allcents/100);  //a.set\_Money(); // 멤버변수 변경 시도 - 2. 참조연산자의 잘 못된 예제  return Money(finaldollars, finalcents);  }  int main()  {  Money p1(10, 40), p2(20), p3;    // 입력  p2.set\_Money();  // 연산자 오버로딩  p3 = p1; // 할당연산자는 자동으로 오버로딩된다.  p3 = p1 + p2; // p3 = operator + (p1, p2); // 공백에 상관이 없다.  // const에 의한 리턴값의 변경 방지  //(p1 + p2).get\_Money();  //(p1 + p2).set\_Money(); // 리턴값의 변경 가능한 예제  //p3 = (p1 + p2).set\_return();  // 교환 법칙 성립  p3 = p2 + p1;  p3 = p2 + 10;  // 출력  cout << "p1: "; p1.get\_Money(); cout << endl;  cout << "p2: "; p2.get\_Money(); cout << endl;  cout << "p3: "; p3.get\_Money(); cout << endl;  return 0;  }  // Money 생성자  Money::Money(int pdollars, int pcents) : dollars(pdollars), cents(pcents) {  }  void Money::set\_Money(){  cout << "dollars & cents=? "; cin >> dollars >> cents;  }  void Money::get\_Money() const {  cout << "dollars & cents= " << dollars << " " << cents << endl;  }  int Money::get\_dollars() const{  return dollars;  }  int Money::get\_cents() const{  return cents;  }  Money Money::set\_return() {  cout << "dollars & cents=? "; cin >> dollars >> cents;  return Money(dollars, cents);  } |

* 멤버 함수로서의 오버로딩

- 연산자 오버로딩을 멤버 함수로 정의한다.

* + - 2항 연산자: 첫 번째 피연산자는 호출 객체이며, 두 번째 피연산자는 호출 함수의 인자이다.
    - 단항 연산자: 첫 번째 피연산자가 호출 객체이며 인자는 없다.

- 클래스의 private 멤버 변수를 직접 접근할 수 있다.

- 멤버 변수를 변경할 수 있으므로 오버로딩 함수 후미에 const 선언하는 것이 좋다.

- 객체지향 정신에 좀 더 부합되는 방식이다.

- 2항 연산자의 교환법칙이 성립하지 않는다.

예제: 멤버 함수로서의 연산자 오버로딩

ex08\_02.cpp

|  |
| --- |
| #include <iostream> // 연산자 오버로딩: 멤버 함수로 오버로딩  using namespace std;  class Money{  public:  Money(int pdollars=0, int pcents=0);  void get\_Money() const;  void set\_Money();  Money set\_return(); // set and return  int get\_dollars() const;  int get\_cents() const;  // 멤버 함수로 연산자 오버로딩  //Money operator+ (Money b)// 1. call by value형 함수  //Money operator+ (const Money &b)// 2. 효율성과 b의 멤버 변수 변경 방지  //const Money operator+ (const Money &b)// 3. 리턴 값의 멤버 변수 변경 방지  //const Money operator+ (const Money &b)// 3. 리턴 값의 멤버 변수 변경 방지  const Money operator+ (const Money &b) const // 4. 자신의 멤버 변수 변경 방지  {  //dollars = 0; // 자신의 멤버 변수를 변경할 수 있다.  int allcents = cents + b.get\_cents();  int finalcents = allcents % 100;  int finaldollars = dollars + b.get\_dollars() + (allcents/100);  return Money(finaldollars, finalcents);  }  private:  int dollars;  int cents;  };  int main()  {  Money p1(10, 40), p2(20, 70), p3;    // 연산자 오버로딩  //p3 = p1 + p2; // p3 = p1.operator+(p2); // 공백에 상관 없다.  p3 = p2 + p1; // p3 = p2.operator+(p1); // 공백에 상관 없다.  // const에 의한 리턴값의 변경 방지  //p3 = (p1+p2).set\_return();  // 교환 법칙 성립하지 않는다.  //p3 = p2 + 10;  //p3 = 10 + p2;  // 출력  cout << "p1: "; p1.get\_Money(); cout << endl;  cout << "p2: "; p2.get\_Money(); cout << endl;  cout << "p3: "; p3.get\_Money(); cout << endl;  return 0;  }  // Money 생성자  Money::Money(int pdollars, int pcents) : dollars(pdollars), cents(pcents) {  }  void Money::set\_Money(){  cout << "dollars & cents=? "; cin >> dollars >> cents;  }  void Money::get\_Money() const {  cout << "dollars & cents= " << dollars << " " << cents << endl;  }  int Money::get\_dollars() const{  return dollars;  }  int Money::get\_cents() const{  return cents;  }  Money Money::set\_return() {  cout << "dollars & cents=? "; cin >> dollars >> cents;  return Money(dollars, cents);  } |

* 기타

- 함수 호출 연산자 ( )는 반드시 멤버 함수로서 오버로딩 되어야 한다//??.잘모르겠음 찾아보기(2021.06.07)

- &&, ||, 콤마는 일반적으로 연산자 오버로딩 하지 않는다.

8.2 프랜드 함수와 자동 형 변환

* 자동 형 변환

- 클래스에 적절한 생성자가 있다면 형 변환은 자동으로 일어난다. 그러나 생성자가 없다면 형 변환은 일어나지 않는다.

* 멤버 함수의 오버로딩과 자동 형 변환

- 자동 형 변환은 호출 객체에서 일어나지 않는다.

- 2항 연산자의 교환 법칙이 성립하지 않기 때문에 대체 방법이 필요하다.

* 프렌드 함수

- 프렌드 함수로 선언되면, 멤버 함수가 아니지만 멤버 함수의 특권을 가지고 있어 멤버 변수의 직접 접근이 가능하다.

- 프렌드 함수의 선언에 ‘friend’ 키워드를 사용하고, 정의부에서는 영역지정 연산자(::)를 사용하지 않는다.

- 연산자 오버로딩에서만 사용 권장

* + - 멤버 함수의 연산자 오버로딩: 클래스의 private 변수를 직접 접근할 수 있다.
    - 전역 함수의 연산자 오버로딩: 자동 형 변환과 교환 법칙이 성립한다.

예제: 프렌드 함수

ex08\_03.cpp

|  |
| --- |
| #include <iostream> // 자동 형 변환과 프랜드 함수  using namespace std;  class Money{  public:  Money(int pdollars=0, int pcents=0);  void get\_Money() const;  void set\_Money();  int get\_dollars() const;  int get\_cents() const;  // 프랜드 함수  **friend const Money operator + (const Money& a, const Money& b);**  **// friend 함수는 클래스의 멤버변수를 접근할 수 있다.**  private:  int dollars;  int cents;  };  **// 프랜드 함수 <-- 일반 함수**  **const Money operator + (const Money& a, const Money& b)**  **{**  **int allcents = a.cents + b.cents;**  **int finalcents = allcents % 100;**  **int finaldollars = a.dollars + b.dollars + (allcents/100);**  **return Money(finaldollars, finalcents);**  **}**  int main()  {  Money p1(10, 40), p2(20, 70), p3;    **// 연산자 오버로딩**  **p3 = p1 + p2;**  **// 교환 법칙**  **p3 = p1 + 10;**  **p3 = 10 + p2;**  // 출력  cout << "p1: "; p1.get\_Money(); cout << endl;  cout << "p2: "; p2.get\_Money(); cout << endl;  cout << "p3: "; p3.get\_Money(); cout << endl;  return 0;  }  // Money 생성자  Money::Money(int pdollars, int pcents) : dollars(pdollars), cents(pcents) {  }  void Money::set\_Money(){  cout << "dollars & cents=? "; cin >> dollars >> cents;  }  void Money::get\_Money() const {  cout << "dollars & cents= " << dollars << " " << cents << endl;  }  int Money::get\_dollars() const{  return dollars;  }  int Money::get\_cents() const{  return cents;  } |

연산자 오버로딩 규칙

1. 최소 하나의 피연산자는 클래스형이어야 한다.
2. =, [ ], ->, ( ) 연산자는 멤버 함수로만 오버로딩 된다.
3. 연산자가 가지고 있는 인자의 수는 변경할 수 없다.
4. 연산자 우선순위를 변경할 수 없다.
5. 도트 연산자(.), 영역지정 연산자(::)는 오버로딩 될 수 없다.
6. 오버로딩 연산자는 디폴트 인자를 갖지 못한다.

8.3 참조와 다른 오버로딩 연산자

* 참조 // 이해 잘 안됨 강의 다시 들어야함.

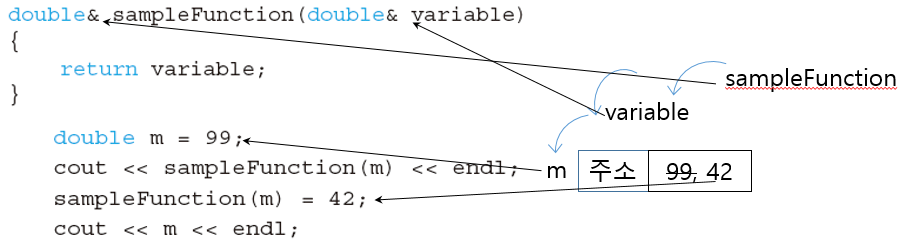


- 저장 위치의 또 다른 이름이다.

- **상수 포인터**이다. 선언시 참조가 결정되고, 포인터 변수처럼 중간에 수정되지 않는다.

* 참조 리턴

- 함수에 의해 리턴되는 객체를 l-vale(할당 연산자의 왼쪽, 값을 저장하는 변수)로 만들고 싶다면 반드시 참조로 리턴해야 한다.



- 함수의 리턴 객체에 어떤 값을 저장할 때 사용한다. 함수의 이름이 변수처럼 사용된다.

- cin, cout의 << 또는 >> 연산자 오버로딩에 사용한다.

예제: 참조와 함수의 반환

ex08\_04.cpp

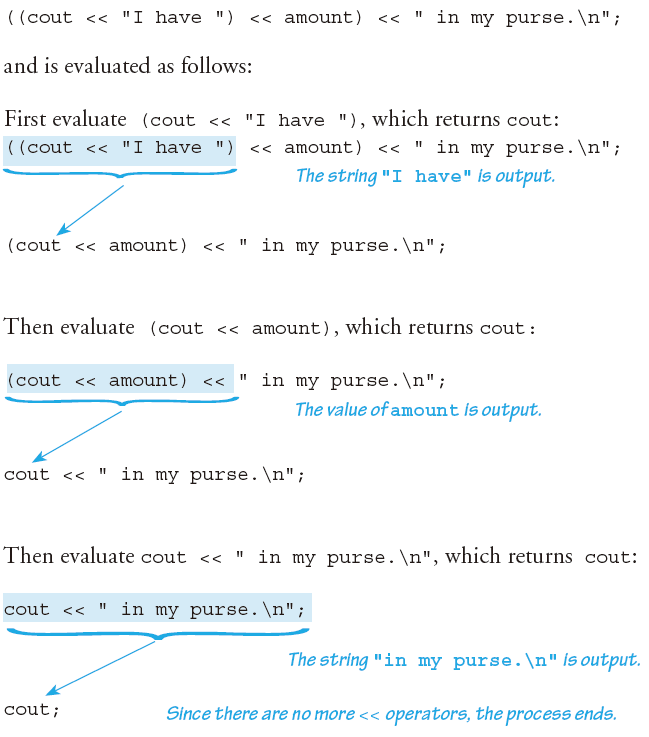
|  |
| --- |
| #include <iostream> // 참조와 함수의 반환  using namespace std;  //int sf(int m){ // call by value //리턴 9, rb=7  //int sf(int &m){ // call by reference //리턴 9, rb=9  int& sf(int &m){ // 참조 리턴 //리턴 9, rb=9  m = 9;  return m;  }  int main()  {  int rb = 7;  int& bob = rb; //int &bob = rb;    cout << rb << " " << bob << endl;  cout << &rb << " " << &bob << endl;  // call by value, call by reference  cout << sf(rb) << endl;  cout << rb << endl;  // 참조 리턴  sf(rb) = 11;  cout << rb << endl; // rb=11  cout << sf(rb) << endl; // 리턴=9  return 0;  } |

* >>와 <<의 오버로딩

- 연산자 >>와 <<는 멤버 함수 형태로 연산자 오버로딩을 못한다.

- >>와 <<는 2항 연산자이다.

원문: 



- cout은 ostream형(클래스)의 사전 정의된 객체이고, cin은 istream형(클래스)의 사전에 정의된 객체이다.

- 리턴값은 istream형 또는 ostream형의 **참조**이어야 한다.

예제: >>와 <<의 오버로딩

ex08\_05.cpp

|  |
| --- |
| #include <iostream> // << >> 멤버 함수 형태로는 연산자 오버로딩 못함.  #include <cmath>  using namespace std;  class Money{  public:  Money(int pdollars=0, int pcents=0);  void get\_Money() const;  void set\_Money();  int get\_dollars() const;  int get\_cents() const;  // 연산자 오버로딩  friend ostream& operator << (ostream& a, const Money& m); // 출력  friend istream& operator >> (istream& b, Money& m); // 입력, const 없다.  private:  int dollars;  int cents;  };  // << 연산자 오버로딩  ostream& operator << (ostream& a, const Money& m)  {  a << "$";  a << m.dollars;  a << ".";  a << m.cents;  return a;  }  // >> 연산자 오버로딩  istream& operator >> (istream& b, Money& m)  {  double amount;  b >> amount;  m.dollars = static\_cast<int>(amount);  int cents = static\_cast<int>(floor(amount\*100+0.5));  m.cents = cents%100;  return b;  }  int main()  {  Money p1(10, 40), p2(50, 90), p3;    // 출력  //p1.get\_Money();  //p2.get\_Money();    // << >> 연산자 오버로딩  // cout << p1 << " " << p2 << endl;  // 출력 형태 $10.40 $50.90  //cout << p1 << " " << p2 << endl;  // 입력  //p3.set\_Money();  // 입력 형태 30.40  //cout << "Enter your money? $" ;  //cin >> p3;  //cout << p3 << endl;    // 입력, 출력  cout << "Enter your money? $" ;  cin >> p1 >> p2;  cout << p1 << " " << p2 << endl;  return 0;  }  // Money 생성자  Money::Money(int pdollars, int pcents) : dollars(pdollars), cents(pcents) {  }  void Money::set\_Money(){  cout << "dollars & cents=? "; cin >> dollars >> cents;  }  void Money::get\_Money() const {  cout << "dollars & cents= " << dollars << " " << cents << endl;  }  int Money::get\_dollars() const{  return dollars;  }  int Money::get\_cents() const{  return cents;  } |

* 증감 연산자의 오버로딩

예제: 전위/후위 증감 연산자

ex08\_06.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  int a = 7, b;  cout << "b = a++;" << endl;  b = a++; // b = a; a = a + 1;  cout << "a=" << a << " b=" << b << endl;  int c = 7, d;  d = ++c; //c = c + 1; d = c;  cout << "d = ++c;" << endl;  cout << "c=" << c << " d=" << d << endl;  // a++; ++c; 동일한 결과  return 0;  } |

연산자 오버로딩

- 단항 연산자로 멤버 함수로 오버로딩한다.

- 전위 증가는 일반적인 멤버 함수의 오버로딩이다.

- 후위 증가는 매개 변수로 int형을 추가한다. 이것은 실제로 매개 변수가 사용되는 것이 아니라, 후위 증가라는 표식으로 사용된다.

예제: 전위/후위 증감 연산자 오버로딩

ex08\_07.cpp

|  |
| --- |
| #include <iostream> // 단항 연산자(++, --), 멤버 함수로 오버로딩  using namespace std;  class Money {  public:  Money(int pdollars = 0, int pcents = 0);  void get\_Money() const;  void set\_Money();  int get\_dollars() const;  int get\_cents() const;  // 전위 / 후위 연산자 오버로딩  Money operator ++(); // 전위 증가  Money operator ++(int); // 후위 증가, operator()++ 안됨  private:  int dollars;  int cents;  };  // 전위 연산자 오버로딩  Money Money::operator ++()  {  ++dollars; // dollars++;  ++cents; // cents++;  return Money(dollars, cents);  }  // 후위 연산자 오버로딩  Money Money::operator ++(int nouse)  {  int temp\_dollars = dollars;  int temp\_cents = cents;  ++dollars;  ++cents;  return Money(temp\_dollars, temp\_cents);  }  int main()  {  Money p1(10, 40), p2(20, 60), p3;  // 연산자 오버로딩  cout << "전위 증가" << endl;  p3 = ++p1; // p3 = p1.operator++();  cout << "p1: " << endl;  p1.get\_Money();  cout << "p3: " << endl;  p3.get\_Money();  cout << "후위 증가" << endl;  p3 = p2++; // p3 = p2.operator()++  cout << "p2: " << endl;  p2.get\_Money();  cout << "p3: " << endl;  p3.get\_Money();  return 0;  }  // Money 생성자  Money::Money(int pdollars, int pcents) : dollars(pdollars), cents(pcents) {  }  void Money::set\_Money() {  cout << "dollars & cents=? "; cin >> dollars >> cents;  }  void Money::get\_Money() const {  cout << "dollars & cents= " << dollars << " " << cents << endl;  }  int Money::get\_dollars() const {  return dollars;  }  int Money::get\_cents() const {  return cents;  } |