프렌드 함수

- 클래스 외부에서 작성된 함수, 멤버가 아님

- 클래스의 멤버 자격 부여, 클래스의 모든 멤버에 대해 접근 가능

- friend키워드로 선언, 개수 제한 없음

friend 함수 필요성

- 클래스의 멤버로 선언하기에는 무리가 있고, 클래스의 모든 멤버를 자유롭게 접근할 수 있는 일부 외부 함수 작성시 ex) 연산자 함수

프렌드 함수가 되는 3가지 유형

- 전역 함수, 타 클래스에 포함된 멤버 함수, 타 클래스에 포함된 멤버 함수 전체

프렌드 함수 선언

- 외부 함수 프렌드로 선언

|  |
| --- |
| friend 외부함수이름(); |

- 특정 클래스의 멤버 함수 프렌드로 선언

|  |
| --- |
| friend 클래스명::멤버함수이름(); |

- 특정 클래스의 모든 멤버 함수 프렌드로 선언

|  |
| --- |
| friend 클래스명(); |

다른 클래스의 private 멤버 함수에 접근하는 경우 friend함수로 오류 해결가능

|  |
| --- |
| class RectManager{  void copy(Rect& dest, Rect& src); }  RectManager::copy(Rect& dest, Rect& src){  dest.width = src.width; dest.height = src.height; } // 두변수는 private라 사용 불가  class Rect{  int width, height;  friend RectManager; } // friend 선언을 통해 오류 해결 |

연산자 중복

- 함수 중복과 함께 다형성 제공

- 기존 연산자에 새로운 기능을 정의

- 높은 프로그램 가독성

연산자 중복의 예시

- 정수 더하기 c = a + b

- 문자열 합치기 c = “C” + “++”

- 색 섞기

- 배열 합치기

연산자 중복의 특징

- C++에 본래 있는 연산자에 대해서만 중복 가능

🡪 모든 연산자가 중복이 가능한 것은 아님

- 피연산자 타입이 다른 새로운 연산 정의

- 본래 연산자의 개수 변경 불가

- 연산 우선 순위 변경은 불가

- 연산자 함수는 함수 형태로 구현

🡪 연산자 함수 (operator function)

🡪 클래스의 멤버 함수 또는 프렌드 함수로 선언

🡪 피연산자에 객체를 동반함

중복이 가능한 연산자

테이블이(가) 표시된 사진

자동 생성된 설명

중복이 불가능한 연산자



연산자 함수 구현

리턴타입 operator 연산자(매개변수리스트)

특징

- 가능하면 클래스 멤버 함수로 작성한다. 새로운 연산자는 클래스와 연동 되므로

- 외부함수로만 구현이 가능한 경우도 있다. (b = int + a)

- 연산자 함수는 디폴트 매개 변수를 가질 수 없다.

클래스에 연산자 함수를 선언하는 경우 – 매개변수 1개

- a + b는 컴파일러에 의해 a . + b로 변하게 됨

🡪 해석 : a 에 있는 연산자 함수 +를 부른다 b를 매개 변수로 하여

|  |
| --- |
| Power a(3, 5), b(4, 6), c;  c = a + b ; // 🡪 a.+(b)  a의 내부  class Power {  int kick;  int punch;  public:  Power operator+ (Power op2); // 클래스 안에 연산자 함수가 있다면 매개변수가 하나면 된다.  } |

프렌드 함수로 연산자 함수를 선언하는 경우 – 매개 변수 2개

|  |
| --- |
| Power a(3, 5), b(4, 6), c;  c = a + b ;  a의 내부  class Power {  int kick;  int punch;  public:  friend Power operator+ (Power op1, Power op2);  } |

+ 연산자 중복 예제 a + b

- 두 객체의 합을 리턴

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power operator + (Power op2);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power Power::operator + (Power op2) {  Power tmp;  tmp.kick = this -> kick + op2.kick;  tmp.punch = this -> punch + op2.punch;  return tmp;  } |

== 연산자 중복 예제 a == b

- 두 객체의 비교 값 결과를 리턴

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  bool operator == (Power op2);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  bool Power::operator == (Power op2) {  if (kick == op2.kick && punch == op2.punch)  return true;  else  return false;  } |

+= 연산자 중복 예제 a += b

- 두 객체를 더해서 앞 객체를 갱신

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power& operator += (Power op2);  };  Power& Power::operator += (Power op2) {  kick = kick + op2.kick  punch = punch + op2.punch;  return \*this; //자기 자신을 리턴 해야 하므로 this 연산자 사용  } |

리턴 타입이 Power&인 이유 : 복사본이 아닌 현재 객체를 참조하기 위해

+ 연산자 중복 예제 2 b = a + int

- 객체에 안의 변수에 값을 더하는 예제

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power operator + (Power op2);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power Power::operator + (int op2) {  Power tmp;  tmp.kick = kick + op;  tmp.punch = punch + op2;  return tmp;  } |

단항 연산자 구현

단항 연산자 : 피연산자가 하나뿐인 연산자

- 종류 : 전위 연산자(++a), 후위 연산자(--a)

전위 ++연산자 구현 ++a

- 객체 안의 변수에 1더하기

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power& operator ++ ();  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power& Power::operator ++ () {  kick++;  punch++;  return \*this;  } |

전위 ! 연산자 구현 !a

- 내부 객체의 변수가 모두 0이면 true 리턴

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  bool operator ! ();  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  bool Power::operator ! () {  if(kick == 0 && punch == 0)  retrun true;  else  return false;  } |

후위 ++연산자 구현 a++

- 객체 안의 변수에 1더하기

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power operator ++ (int x );  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power Power::operator ++ () {  Power tmp = \*this;  kick++;  punch++;  return tmp; //증가 이전의 값을 저장 받아 리턴  } |

friend 함수를 이용한 연산자 함수 예제

- 외부 함수를 이용해야 하는 경우 이용 연산자 . (매개변수, 매개변수) 형식으로 변환

+ 연산자 중복 예제 3 b = int + a

- b = + . (int, a) 으로 변환하여 연산자 함수 호출 (외부로 호출하여 friend 함수로 호출)

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  friend Power operator ++ ();  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power operator + (int po1, Power op2) {  Power tmp;  tmp.kick = kick + op;  tmp.punch = punch + op2;  return tmp;  } |

+ 연산자 중복 예제 4 a + b

- a + b를 + . (a, b)로 변환하여 연산 friend 함수 이용

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  friend Power operator + (Power op2);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power operator + (Power op1, Power op2) {  Power tmp;  tmp.kick = op1.kick + op2.kick;  tmp.punch = op1.punch + op2.punch;  return tmp;  } |

단항 연산자 구현 2

- ++ . (a, 0)[후위] 혹은 ++ . (0, a)[전위]

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  friend Power& operator ++ (Power& op);  friend Power operator ++ (Power& op, int x);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power& operator ++ (Power& op) { // 전위 ++ 연산자  op.kick++;  op.punch++;  return op;  }  Power operator ++ (Power& op, int x) { // 후위 ++ 연산자  Power tmp = op; // 변경 이전의 op저장  op.kick++;  op.punch++;  return tmp; // 변경 이전의 op리턴  } |

참조를 리턴 하는 << 연산자 작성

- 객체에 정수를 더하는 연산자 a << int

|  |
| --- |
| class Power {  int kick;  int punch;  public:  Power(int kick=0, int punch=0) {  This->kick = kick; this->punch = punch;  }  void show();  Power& operator << (Power op2);  };  void Power::show() {  cout << “kick=” << kick << ‘,’ << “punch=” << punch << endl;  }  Power& Power::operator << (int n) {  kick += n;  punch += n;  return \*this;  } |

상속

C++에서의 상속관계 정의

- 객체 사이에는 상속 관계 없음

- 기본 클래스의 속성과 기능을 파생 클래스에 물려주는 것

기본 클래스(base class)

- 상속해주는 클래스

파생 클래스(derived class)

- 상속받는 클래스 (자식 클래스)

- 기본 클래스의 속성과 기능을 물려받고 자신 만의 속성과 기능을 추가하여 작성

상속의 활용

- 기본 클래스에서 파생 클래스로 갈수록 클래스의 개념이 구체화

- 다중 상속을 통한 클래스의 재활용성을 높임

상속의 장점 및 목적

1. 간결한 클래스 작성

- 기본 클래스 기능을 물려받아 파생 클래스를 간결하게 작성

2. 클래스 간의 계층적 분류 및 관리의 용이함

- 상속은 클래스들의 구조적 관계 파악 용이

3. 클래스 재사용과 확장을 통한 소프트웨어 생산성 향상

- 빠른 소프트웨어 생산 필요

- 기존에 작성한 클래스의 재사용 – 상속을 받아 새로운 기능을 확장

- 앞으로 있을 상속에 대비한 클래스의 객체 지향적 설계 필요

상속 선언

|  |
| --- |
| class Student : public Person {  // Person을 상속받는 Student 생성  };  class StudentWorker : public Student {  // Student을 상속받는 StudentWorker 생성 -> person도 같이 물려 받음  }; |

- 파생 클래스는 기본 클래스의 메서드와 변수를 자신의 것 같이 사용할 수 있다.

업 캐스팅 (up-casting)

- 파생 클래스 포인터가 기본 클래스 포인터에 치환되는 것

파생 클래스와 기본 클래스 모두 사용이 가능한 상태에서 기본 클래스만 사용하는 것

다운 캐스팅 (down-casting)

- 기본 클래스의 포인터가 파생 클래스의 포인터에 치환되는 것

기본 클래스만 사용이 가능한 상태에서 파생 클래스와 기본 클래스 모두 사용하는 것

|  |
| --- |
| ColorPoint cp;  ColorPoint \*pDer;  Point\* pBase =&cp; // 업캐스팅  pDer = (ColorPoint \*)pBase; // 다운 캐스팅 |

다운 캐스팅의 잘못된 예

|  |
| --- |
| ColorPoint \*pDer;  Point \*pBase, po;  pBase = &po;  pDer = (ColorPoint \*)pBase; // pDer에는 할당 되지 않아서 ColorPoint사용 불가 컴파일러 오류 x |

접근 지정자

Private 멤버

- 선언된 클래스 내에서만 접근 가능

- 파생 클래스에서도 기본 클래스의 private 멤버 직접 접근 불가

Public 멤버

- 선언된 클래스 외부 어떤 클래스, 모든 외부 함수에 접근 허용

- 파생 클래스에서 기본 클래스의 public 멤버 접근 가능

Protected 멤버

- 선언된 클래스에서 접근 가능

- 파생 클래스에서만 접근 허용

- 파생 클래스가 아닌 다른 클래스나 외부 함수에서는 접근 불가

상속 관계의 생성자와 소멸자

파생 클래스 객체가 생성될 때

- 파생 클래스와 기본 클래스 생성자 모두 실행됨

실행 순서

1 기본 클래스 생성자 실행

2 파생 클래스 생성자 실행

파생 클래스 객체가 소멸될 때

- 파생 클래스와 기본 클래스 소멸자 모두 실행됨

실행 순서

1 파생 클래스 소멸자 실행

2 기본 클래스 소멸자 실행

파생 클래스의 생성자에서 기본클래스의 생성자를 명시하지 않은 경우

|  |
| --- |
| class A {  public:  A(int x) { } ;  };  class B : public A {  public:  B() { }; // A의 기본 생성자를 불러올 수 없어서 오류 발생  }; |

파생클래스의 생성자에서 기본 클래스의 생성자 명시적으로 지정하는 방법

|  |
| --- |
| class A {  public:  A(int x) { } ;  };  class B : public A {  public:  B(int x) : A(x + 3){ }; // B의 매개 변수도 사용 가능  }; |

상속 지정

- 상속 선언 시 public, private, protected의 3가지 중 하나 지정

- 기본 클래스의 멤버 접근 속성을 어떻게 계승할지 지정

Public – 기본 클래스의 protected, public 멤버 속성을 그대로 계승

Private - 기본 클래스의 protected, public 멤버 속성을 private으로 계승

Protected - 기본 클래스의 protected, public 멤버 속성을 protected로 계승

상속 중첩시에 접근 지정 사례

|  |
| --- |
| class Base {  int a;  protected:  void setA(int a) { this -> a = a; }  public:  void showA() { cout << a; }  };  class Derived : private Base{  int b;  protected:  void setB (int b) { this -> b = b; }  public:  void showB() {  setA(5);  showA(); // 같은 클래스 안에 있다고 판단하고 사용 가능  cout << b }  };  class Derived : private Derived {  int c;  protected:  void showAB(int x) {  setA(x); //error  showA(); //error // 상속 중첩시에는 불가능  setB(x); }  }; |

다중 상속

- 하나의 클래스가 여러 클래스에서 상속 받는 것

|  |
| --- |
| class A{  };  class B{  };  class C : public A, public B {  // 다중 상속  }; |

다중 상속의 문제점

- 클래스가 이중으로 삽입되는 문제점 -> 가상 상속으로 문제 해결 가능

|  |
| --- |
| class A{  };  class B: public A {  };  class C : public A {  };  class D : public B, public C { // 컴파일 오류  }; |

가상 상속

- 파생 클래스의 선언 문에서 기본 클래스 앞에 virtual로 선언

- 파생 클래스의 객체가 생성될 때 기본 클래스의 멤버는 오직 한 번만 생성

|  |
| --- |
| class A{  };  class B: virtual public A {  };  class C : virtual public A {  };  class D : public B, public C {  }; |

상속 관계 함수 중복

|  |
| --- |
| class Base {  public:  void f() {}  };  class Derived : public Base{  public:  void f() {}  };  Int main() {  Derived d, \*pDer;  pDer = & d;  pDer -> f(); // Derived f() 호출  Base\* pBase;  pBase = pDer; // 업캐스팅  pBase -> f (); // Base f() 호출  } |

가상 함수 (virtual function)

- virtual 키워드로 선언된 함수

Virtual

- 동적 바인딩 지시어

- 컴파일러에게 함수에 대한 호출 바인딩을 실행 시간까지 미루도록 지시

함수 오버라이딩 (function overriding)

- 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언

- 기본 클래스의 가상 함수 존재감 상실시킴

- 파생 클래스에서 오버라이딩한 함수가 호출되도록 동적 바인딩

- 함수 재정의라고도 부름

- 다형성의 한 부류

오버 로딩과 오버 라이딩

오버 로딩 – 함수의 중복으로 함수가 덮어 쓰임

오버 라이딩 – 가상 함수 선언으로 기본 클래스의 함수가 존재감을 아예 잃음

텍스트이(가) 표시된 사진

자동 생성된 설명

함수 오버라이딩

|  |
| --- |
| class Base {  public:  virtual void f() {}  };  class Derived : public Base{  public:  virtual void f() {}  };  Int main() {  Derived d, \*pDer;  pDer = & d;  pDer -> f(); // Derived f() 호출  Base\* pBase;  pBase = pDer; // 업캐스팅  pBase -> f (); // Derived f() 호출  } |

기본 클래스에 가상 함수를 만드는 목적

- 파생 클래스들이 자신의 목적에 맞게 가상 함수를 재정의 함

- 상속받는 파생 클래스에서 구현할 함수 인터페이스 제공

- 파생 클래스의 다형성(polymorphism) 실현 : 하나의 인터페이스에 대해 서로 다른 모양의 구현

동적 바인딩 (dynamic binding)

- 기본 클래스에 대한 포인터로 가상 함수를 호출하는 경우, 객체 내에 오버라이딩한 파생 클래스의 함수를 찾아 실행

- 실행 중에 이루어짐

- 런타임 바인딩(run-time binding), 늦은 타임(late binding) 바인딩으로도 불림

- 파생 클래스에서 재정의한 가상 함수의 호출을 보장받는 선언

C++ 오버 라이딩 특징

- 오버라이딩의 성공 조건: 가상 함수 이름, 매개 변수 타입과 개수, 리턴 타입이 모두 일치

- 오버라이딩시 virtual 지시어 생략 가능

-> 가상 함수의 virtual 지시어는 상속됨, 파생 클래스에서 virtual 생략 가능

- 가상 함수의 접근 지정

-> private, protected, public 중 자유롭게 지정

상속이 반복되는 경우 가상 함수 호출

|  |
| --- |
| class Base {  public:  virtual void f() {}  };  class Derived : public Base{  public:  void f() {}  };  class GrandDerived : public Derived{  public:  void f() {} // 여기서 오버라이딩된 함수가 호출 됨  }; |

범위 지정 연산자(::)

- 정적 바인딩 지시

- 기본클래스::가상함수() 형태로 기본 클래스의 가상 함수를 정적 바인딩으로 호출

- 메인에서 사용하여 해당 가상 함수를 정적 바인딩으로 호출이 가능

- 기본 ::함수() 사용시 외부 함수 호출

가상 소멸자

- 소멸자를 virtual 키워드로 선언

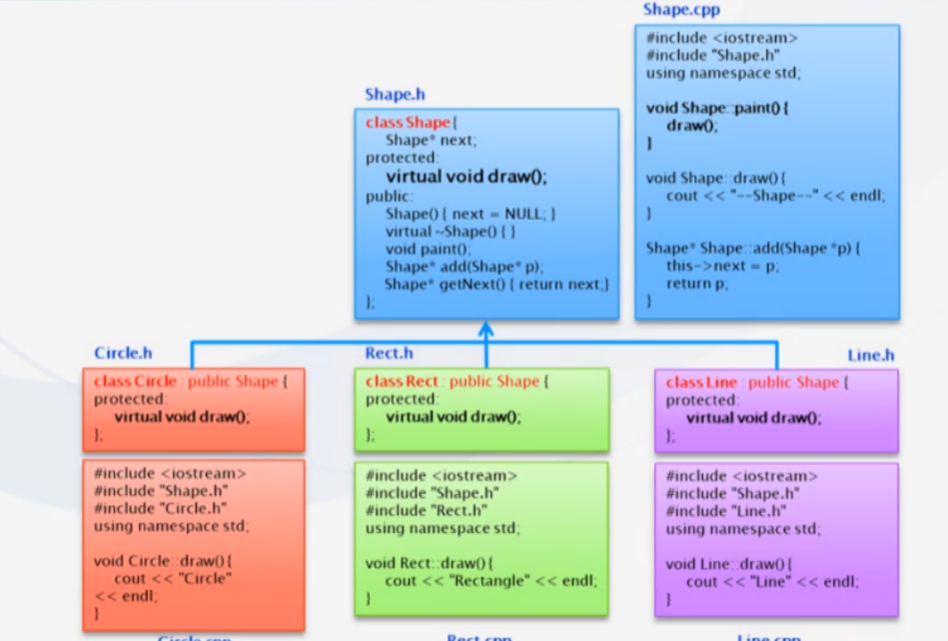
- 소멸자 자동 호출 시 동적 바인딩 발생

|  |
| --- |
| class Base {  public:  ~Base();  };  class Derived : public Base{  public:  ~Derived();  };  int main() {  Base \*p = new Derived();  delet p; // ~Base() 소멸자만 실행  } |
| class Base {  public:  virtual ~Base();  };  class Derived : public Base{  public:  virtual ~Derived();  };  int main() {  Base \*p = new Derived();  delet p; // ~Derived() 소멸자, ~Base() 소멸자 실행  } |

테이블이(가) 표시된 사진

자동 생성된 설명

가상 함수와 오버 로딩 사용 사례



텍스트이(가) 표시된 사진

자동 생성된 설명

순수 가상 함수

기본 클래스의 가상 함수 목적

- 파생 클래스에서 재정의할 함수를 알려주는 역할 -실행할 목적이 아님

순수 가상 함수 (pure virtual function)

- 함수의 코드가 없고 선언만 있는 가상 멤버 함수

선언 방법

멤버 함수의 원형=0;으로 선언

virtual void f()=0;

추상 클래스

- 최소한 하나의 순수 가상 함수를 가진 클래스

추상 클래스의 특징

- 온전한 클래스가 아니므로 객체 생성 불가능

- 추상 클래스의 포인터는 선언 가능

추상 클래스의 목적

- 추상 클래스의 인스턴스를 생성할 목적이 아님

- 상속에서 기본 클래스의 역할을 하기 위함

-> 순수 가상 함수를 통해 파생클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할

-> 추상 클래스의 모든 멤버 함수를 순수 가상 함수로 선언할 필요 없음

추상 클래스의 상속

- 추상 클래스를 단순 상속하면 자동 추상 클래스

추상 클래스의 구현

- 추상 클래스를 상속받아 순수 가상함수를 오버라이딩하여 구현하면 추상클래스가 아님

일반화와 템플릿

일반화 (제네릭 generic)

- 함수나 클래스를 일반화시키고, 매개 변수 타입을 지정하여 틀에서 찍어 내듯이 함수나 클래스 코드를 생산하는 기법

템플릿 (template)

- 함수나 클래스를 일반화하는 C++ 도구

- template 키워드로 함수나 클래스 선언

-> 변수나 매개 변수의 타입만 다르고, 코드 부분이 동일한 함수를 일반화시킴

- 제네릭 타입

-> 일반화를 위한 데이터 타입

|  |
| --- |
| template <class T> // 템플릿 선언  template <class T1, class T2, class T3> // 여러 개 선언 |

- 함수를 호출할 때 사용할 타입 지정(구체화)

구체화 (specialization)

- 템플릿의 제네릭 타입에 구체적인 타입 지정

- 템플릿 함수로부터 구체화된 함수의 소스 코드 생성

- 컴파일러가 컴파일할 때 자동으로 시켜줌

- 구체적 타입 지정 시 동일한 제네릭 타입에는 동일한 타입이 들어가야 한다

템플릿 장점

- 함수 코드의 재사용: 높은 소프트웨어 생산성과 유동성

템플릿 단점

- 포팅에 취약: 컴파일러에 따라 지원하지 않을 수 있음

- 컴파일 오류 메시지 빈약 디버깅에 많은 어려움

제네릭 프로그래밍 (generic programming)

- 일반화 프로그래밍이라고도 부름

- 제네릭 함수나 제네릭 클래스를 활용하는 프로그래밍 기법

- C++에서 STL(Standard Template Library) 제공, 활용

- 다른 언어에서도 보편화 되는 추세

템플릿 함수의 문제점 예제

|  |
| --- |
| template <class T>  void print(T array [], iny n) { // 템플릿 함수  for(int i =0; i < n; i ++)  cout << array << ‘\t’;  cout << endl;  }  void print(char array [], iny n) { // 중복 함수  for(int i =0; i < n; i ++)  cout << (int)array << ‘\t’;  cout << endl;  }  Int main() {  char c[5] = {1, 2, 3, 4, 5};  print(c, 5); // 템플릿 함수보다 중복 함수가 더 우선적으로 호출  } |

제네릭 클래스

제네릭 클래스 선언

|  |
| --- |
| template <class T>  class A {}; |

제네릭 클래스 구현 (구현부)

|  |
| --- |
| template <class T>  void A<T>::a(T n) { };  template <class T> T A<T>::a2() { }; |

제네릭 클래스 구체화

|  |
| --- |
| A<int> a1;  A<double> a2; |

- 선언을 할 때 미리 다룰 타입 선언(포인터도 가능)

C++ 템플릿 라이브러리

STL (Standard Template Library)

- 표준 템플릿 라이브러리

-> C++ 표준 라이브러리 중 하나

- 많은 제네릭 클래스와 제네릭 함수 포함

-> 개발자는 이들을 이용하여 쉽게 응용 프로그램 작성

STL 구성

컨테이너(container) – 템플릿 클래스

- 데이터를 담아두는 자료 구조를 표현한 클래스

- 리스트, 큐, 스택, 맵, 셋, 벡터

반복자(iterator) – 컨테이너 원소에 대한 포인터

- 컨테이너 원소들을 순회하면서 접근하기 위해 만들어진 컨테이너 원소에 대한 포인터

알고리즘 – 템플릿 함수

- 컨테이너 원소에 대한 복사, 검색, 삭제, 정렬 등의 기능을 구현한 템플릿 함수

- 컨테이너의 멤버함수 아님

테이블이(가) 표시된 사진

자동 생성된 설명

STL관련 헤더 파일과 이름 공간

헤더 파일

- 해당 클래스가 선언된 헤더 파일 include ex) include<vector>

- 알고리즘은 함수에 상관 없이 #include <algorithm>

이름 공간

- STL이 선언된 namespace는 std

Vector 컨테이너

- 가변 길이 배열을 구현한 제네릭 클래스

-> 개발자가 벡터 길이에 대한 고민할 필요 없음

- 원소의 저장, 삭제, 검색 등 다양한 멤버 함수 지원

- 벡터에 저장된 원소는 인덱스로 접근 가능

Vector의 멤버 함수와 연산자 함수

push\_back(element) – 벡터의 마지막에 element 추가

at(int index) – index 위치의 원소에 대한 참조 리턴

begin() – 벡터의 첫번째 원소에 대한 참조 리턴

end() – 벡터의 끝(마지막 원소 다음)을 가리키는 참조 리턴

empty() – 벡터가 비어 있다면 true 리턴

erase(iterator it) – 벡터에서 it가 가리키는 원소 삭제, 삭제 후 자동으로 벡터 조정

insert(iterator it, element) – 벡터 내 it 위치에 element 삽입

size() – 벡터에 들어 있는 원소의 개수 리턴

capacity() – 할당된 저장 공간 수 리턴

operator[]() – 지정된 원소에 대한 참조 리턴

operator=() – 이 벡터를 다른 벡터에 치환(복사)

iterator

- 반복자라고도 부름

- 컨테이너 원소를 가리키는 포인터

- 구체적인 컨테이너를 지정하여 반복자 변수 생성

|  |
| --- |
| 선언  vector<int>::iterator it;  it = v.begin();  증가  it++; // 다음원소를 가리킴  원소 읽기  int n = \*it;  원소 쓰기  n = n\*2;  \*it = n;  원소 삭제  it = v.erase(it); // 삭제 후 다음 원소에 대한 포인터 리턴  끝으로 옮기기  It = v.end(); |

Iterator 하나씩 순서대로 리턴

for(it = v.begin(); it != v.end(); it++)

map 컨테이너

- 키와 값의 쌍으로 원소를 저장하는 제네릭 컨테이너

-> 동일한 키의 값을 가지고 있다면 오류 발생

- 키로 값을 검색

make\_pair(key, value) – ‘키’와 ‘값’을 가진 pair 객체 생성

Map의 멤버 함수와 연산자 함수

insert(pair<> &element) – 맵에 ‘키’와 ‘값’으로 구성된 pair 객체 element 삽입

at(key\_type& key) – 맵에서 ‘키’에 해당하는 ‘값’리턴

begin() – 맵의 첫번째 원소에 대한 참조 리턴

end() – 맵의 끝(마지막 원소 다음)을 가리키는 참조 리턴

empty() – 맵이 비어 있다면 true 리턴

erase(iterator it) – 맵에서 it가 가리키는 원소 삭제

size() – 맵에 들어 있는 원소의 개수 리턴

operator[key\_type& key]() – 맵에서 ‘키’에 해당하는 원소를 찾아 ‘값’리턴

operator=() – 맵 치환(복사)

map의 원소 추가 방법 (2가지)

|  |
| --- |
| map.insert(make\_pair(key, value));  map[key] = value; |

알고리즘 함수

- 템플릿 함수, 전역 함수 (STL 컨테이너 클래스의 멤버 함수가 아님)

- iterator와 함께 작동

- #include <algorithm> 선언으로 사용

sort() 함수

- 특정 범위의 원소 정렬

첫번째 매개 변수 : 정렬범위의 시작 원소 주소

두번째 매개 변수 : 정렬범위의 마지막 원소 다음 주소

|  |
| --- |
| sort(v.begin(), v.begin()+3); // begin() ~ begin()+2  sort(v.begin()+2, v.begin()+5); // begin()+2 ~ begin()+4  sort(v.begin(), v.end()); // 전체 |

auto

- 컴파일러에게 변수선언문에서 추론하여 타입을 자동 선언하도록 지시

- C++11이전에는 스택에 할당되는 지역 변수 선언 키워드

장점 – 복잡한 변수 선언을 간소하게, 긴 타입 선언 시 오타 줄임

|  |
| --- |
| auto pi = 3.14; // double  auto n = 3; // int  auto \*p = &n; // int\*  int & ref = n;  auto & ref2 = ref; // int& |

리턴 타입 추론과 iterator 추론도 가능

|  |
| --- |
| for(auto it = v.begin(); it != v.end(); it++) |

람다

- 람다 대수에서 람다식은 수학 함수를 단순하게 표현하는 기법

f(x, y) = x + y 🡪 (x, y) -> x + y

- 이름 없는 익명의 함수를 만드는 기능으로 C++11에 도입

람다식 구성

캡쳐 리스트 : 람다식에서 사용하고자 하는 함수 바깥의 변수 목록

매개변수 리스트 : 보통의 함수의 매개 변수 리스트와 동일

리턴 타입

함수 바디 : 람다식의 함수 코드

|  |
| --- |
| [캡쳐 리스트] (매개변수 리스트) -> 리턴 타입 {함수 바디};  Ex) []{int x, int y} -> int { return x + y } (2, 3); 🡪 5 리턴 |

auto로 람다식 저장 및 호출

|  |
| --- |
| auto love = [](string a, string b) -> string { return a + b };  love(“a”, “b”); |

for\_each()

첫번째 매개 변수 : 검색범위의 시작 원소 주소

두번째 매개 변수 : 검색범위의 마지막 원소 다음 주소

세번째 매개 변수 : 호출할 함수

|  |
| --- |
| void print(int n) { cout << n << “ ”; }  Vector<int> v = {1, 2, 3, 4, 5};  for\_each(v.begin(), v.end(), print);  람다식 변형 -> for\_each(v.begin(), v.end(), [](int n) { cout << n << “ ”; }); |