예제 9-6(실습) 추상 클래스 구현 연습

1

다음 추상 클래스 Calculator를 상속받아 GoodCalc 클래스를 구현하라.

```
class Calculator {
public:
    virtual int add(int a, int b) = 0; // 두 정수의 합 리턴
    virtual int subtract(int a, int b) = 0; // 두 정수의 차 리턴
    virtual double average(int a [], int size) = 0; // 배열 a의 평균 리턴. size는 배열의 크기
};
```

```
#include <iostream>
using namespace std;

// 이 곳에 Calculator 클래스 코드 필요

class GoodCalc : public Calculator {
public:
    int add(int a, int b) { return a + b; }
    int subtract(int a, int b) { return a - b; }
    double average(int a [], int size) {
        double sum = 0;
        for(int i=0; i<size; i++)
            sum += a[i];
        return sum/size;
    }
};
```

순수 가상 함수 구현

```
int main() {
    int a[] = {1,2,3,4,5};
    Calculator *p = new GoodCalc();
    cout << p->add(2, 3) << endl;
    cout << p->subtract(2, 3) << endl;
    cout << p->average(a, 5) << endl;
    delete p;
}
```

```
5
-1
3
```

예제 9-7(실습) 추상 클래스를 상속받는 파생 클래스 구현 연습

다음 코드와 실행 결과를 참고하여 추상 클래스 Calculator를 상속받는 Adder와 Subtractor 클래스를 구현하라.

```
#include <iostream>
using namespace std;

class Calculator {
    void input() {
        cout << "정수 2 개를 입력하세요>> ";
        cin >> a >> b;
    }
protected:
    int a, b;
    virtual int calc(int a, int b) = 0; // 두 정수의 합 리턴
public:
    void run() {
        input();
        cout << "계산된 값은 " << calc(a, b) << endl;
    }
};

int main() {
    Adder adder;
    Subtractor subtractor;
    adder.run();
    subtractor.run();
}
```

adder.run()에 의한 실행 결과

subtractor.run()에 의한 실행 결과

```
정수 2 개를 입력하세요>> 5 3
계산된 값은 8
정수 2 개를 입력하세요>> 5 3
계산된 값은 2
```

10

템플릿과 표준 템플릿 라이브러리(STL)

학습 목표

1. 일반화와 템플릿의 개념과 목적을 이해한다.
2. 템플릿으로부터 구체화의 과정을 이해한다.
3. 템플릿 함수와 템플릿 클래스를 작성하고 활용할 수 있다.
4. C++ 표준 템플릿 라이브러리(STL)에 대해 이해한다.
5. STL의 vector, map 컨테이너를 이해하고 활용할 수 있다.
6. STL의 iterator와 알고리즘 함수에 대해 이해하고 간단히 활용할 수 있다.
7. auto로 변수를 쉽게 선언하는 것을 알고 활용할 수 있다.
8. 람다식의 개념을 알고 간단한 람다식을 작성하고, 호출할 수 있다.

함수 중복의 약점 – 중복 함수의 코드 중복


```
#include <iostream>
using namespace std;

void myswap(int& a, int& b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
void myswap(double & a, double & b) {
    double tmp;
    tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a=4, b=5;
    myswap(a, b); // myswap(int& a, int& b)  호출
    cout << a << '₩t' << b << endl;

    double c=0.3, d=12.5;
    myswap(c, d); // myswap(double& a, double& b) 호출
    cout << c << '₩t' << d << endl;
}
```

두 함수는 매개 변수만 다르고 나머지 코드는 동일함

동일한 코드 중복 작성

```
5       4
12.5    0.3
```

일반화와 템플릿

- 제네릭(generic) 또는 일반화
  - 함수나 클래스를 일반화시키고, 매개 변수 타입을 지정하여 틀에서 찍어 내듯이 함수나 클래스 코드를 생산하는 기법
- 템플릿
  - 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

- 템플릿 선언

```
template <class T> 또는
template <typename T>

3 개의 제네릭 타입을 가진 템플릿 선언
template <class T1, class T2, class T3>
```

템플릿을 선언하는 키워드

제네릭 타입을 선언하는 키워드

제네릭 타입 T 선언

```
template <class T>
void myswap (T & a, T & b) {
   T tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

템플릿을 이용한 제네릭 함수 myswap

중복 함수들로부터 템플릿 만들기 사례


```
void myswap(int & a, int & b) {
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

```
void myswap (double & a, double & b) {
   double tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

중복 함수들

제네릭 함수
만들기(일반화)

템플릿을 선언하는 키워드

제네릭 타입을 선언하는 키워드

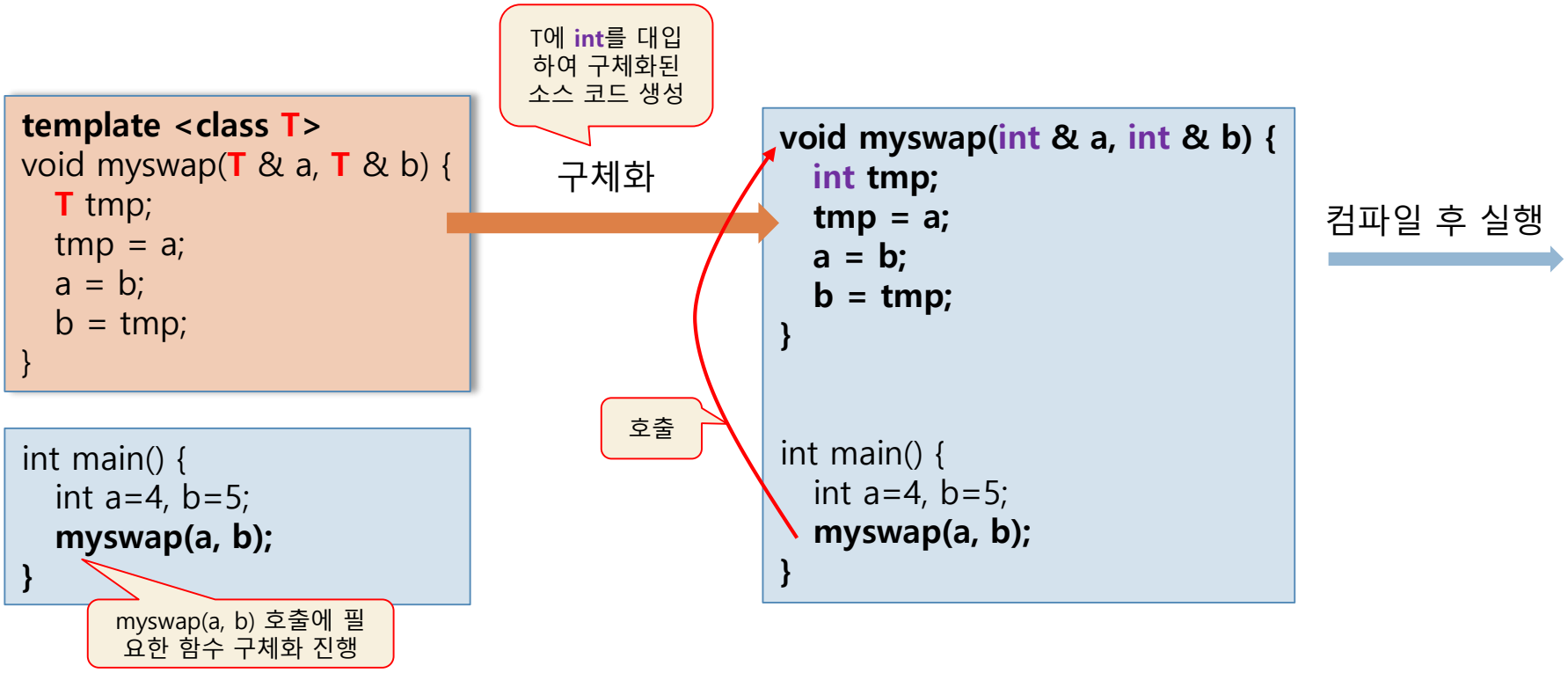
제네릭 타입 T 선언

```
template <class T>
void myswap (T & a, T & b) {
   T tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

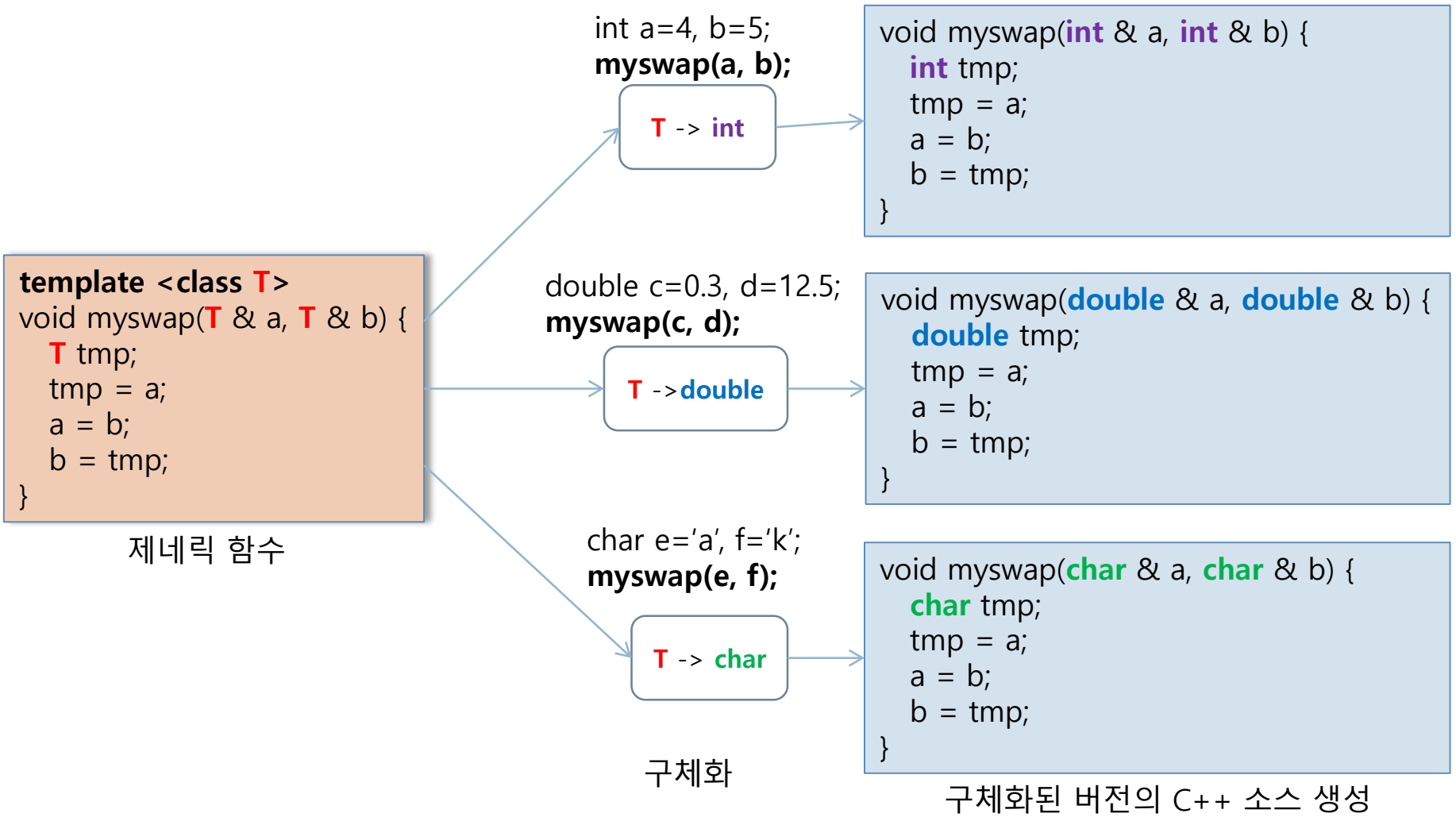
템플릿을 이용한
제네릭 함수

템플릿으로부터의 구체화

- 구체화(specialization)
  - 템플릿의 제네릭 타입에 구체적인 타입 지정
    - 템플릿 함수로부터 구체화된 함수의 소스 코드 생성



제네릭 함수로부터 구체화된 함수 생성 사례



예제 10–1 제네릭 myswap() 함수 만들기


```
#include <iostream>
using namespace std;

class Circle {
    int radius;
public:
    Circle(int radius=1) { this->radius = radius; }
    int getRadius() { return radius; }
};

template <class T>
void myswap(T & a, T & b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int a=4, b=5;
    myswap(a, b);
    cout << "a=" << a << ", " << "b=" << b << endl;

    double c=0.3, d=12.5;
    myswap(c, d);
    cout << "c=" << c << ", " << "d=" << d << endl;

    Circle donut(5), pizza(20);
    myswap(donut, pizza);
    cout << "donut반지름=" << donut.getRadius() << ", ";
    cout << "pizza반지름=" << pizza.getRadius()<< endl;
}
```

myswap(int& a, int& b)
함수 구체화 및 호출

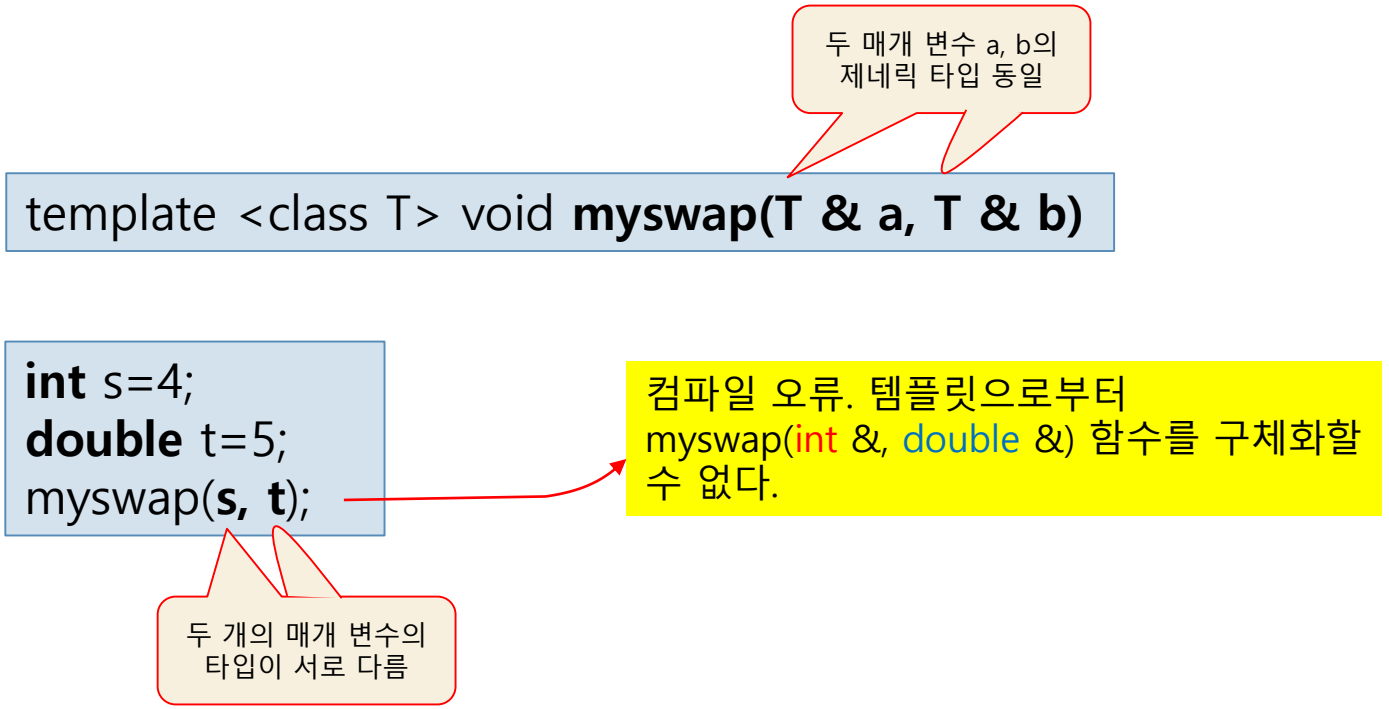
myswap(double& a, double& b)
함수 구체화 및 호출

myswap(Circle& a, Circle& b)
함수 구체화 및 호출

```
a=5, b=4
c=12.5, d=0.3
donut반지름=20, pizza반지름=5
```

구체화 오류

- 제네릭 타입에 구체적인 타입 지정 시 주의



템플릿 장점과 제네릭 프로그래밍

- 템플릿 장점
  - 함수 코드의 재사용
    - 높은 소프트웨어의 생산성과 유용성
- 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
  - 보편화 추세
    - Java, C# 등 많은 언어에서 활용

예제 10–2 큰 값을 리턴하는 bigger() 함수 만들기 연습

두 값을 매개 변수로 받아 큰 값을 리턴하는 제네릭 함수 bigger()를 작성하라.

```
#include <iostream>
using namespace std;









int main() {
   int a=20, b=50;
   char c='a', d='z';
   cout << "bigger(20, 50)의 결과는 " << bigger(a, b) << endl;
   cout << "bigger('a', 'z')의 결과는 " << bigger(c, d) << endl;
}
```

```
bigger(20, 50)의 결과는 50
bigger('a', 'z')의 결과는 z
```

예제 10–3 배열의 합을 구하여 리턴하는 제네릭 add() 함수 만들기 연습

배열과 크기를 매개 변수로 받아 합을 구하여 리턴하는 제네릭 함수 add()를 작성하라.

```
#include <iostream>
using namespace std;

template <class T>
T add(T data [], int n) { //  배열 data에서 n개의 원소를 합한 결과를 리턴
   T sum = 0;
   for(int i=0;  i<n; i++) {
      sum += data[i];
   }
   return sum; // sum와 타입과 리턴 타입이 모두 T로 선언되어 있음
}

int main() {
   int x[] = {1,2,3,4,5};
   double d[] = {1.2, 2.3, 3.4, 4.5, 5.6, 6.7};

   cout << "sum of x[] = " << add(x, 5) << endl; // 배열 x와 원소 5개의 합을 계산
   cout << "sum of d[] = " << add(d, 6) << endl; // 배열 d와 원소 6개의 합을 계산
}
```

```
sum of x[] = 15
sum of d[] = 23.7
```

예제 10-4 배열을 복사하는 제네릭 함수 mcopy() 함수 만들기 연습

두 개의 배열을 매개 변수로 받아 배열을 복사하는 제네릭 mcopy() 함수를 작성하라.

```
#include <iostream>
using namespace std;

// 두 개의 제네릭 타입 T1, T2를 가지는 copy()의 템플릿
template <class T1, class T2>
void mcopy(T1 src [], T2 dest [], int n) { // src[]의 n개 원소를 dest[]에 복사하는 함수
    for(int i=0; i<n; i++)
        dest[i] = src[i]; // T1 타입의 값을 T2 타입으로 변환한다.
}

int main() {
    int x[] = {1,2,3,4,5};
    double d[5];
    char c[5] = {'H', 'e', 'l', 'l', 'o'}, e[5];

    mcopy(x, d, 5); // int x[]의 원소 5개를 double d[]에 복사
    mcopy(c, e, 5); // char c[]의 원소 5개를 char e[]에 복사

    for(int i=0; i<5; i++) cout << d[i] << ' '; // d[] 출력
    cout << endl;
    for(int i=0; i<5; i++) cout << e[i] << ' '; // e[] 출력
    cout << endl;
}
```

mcopy()의 T1은 int로, T2는 double로 구체화

mcopy()의 T1, T2 모두 char로 구체화

```
1 2 3 4 5
H e l l o
```

배열을 출력하는 print() 템플릿 함수의 문제점

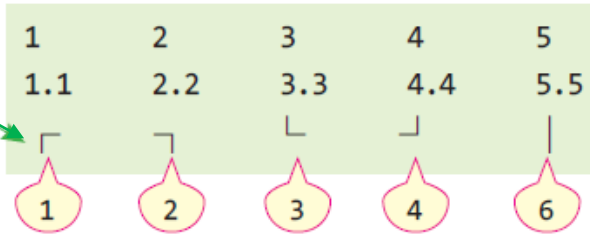

```
#include <iostream>
using namespace std;

template <class T>
void print(T array [], int n) {
    for(int i=0; i<n; i++)
        cout << array[i] << '₩t';
    cout << endl;
}

int main() {
    int x[] = {1,2,3,4,5};
    double d[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    print(x, 5);
    print(d, 5);

    char c[5] = {1, 2, 3, 4, 5};
    print(c, 5);
}
```

T가 char로 구체화되는 경우, 정수 1, 2, 3, 4, 5에 대한 그래픽 문자 출력

print() 템플릿의 T가 int 타입으로 구체화

print() 템플릿의 T가 char 타입으로 구체화

char로 구체화되면 숫자대신 문자가 출력되는 문제 발생!



예제 10-5 템플릿 함수보다 중복 함수가 우선

```
#include <iostream>
using namespace std;

template <class T>
void print(T array [], int n) {
    for(int i=0; i<n; i++)
        cout << array[i] << '₩t';
    cout << endl;
}

void print(char array [], int n) { // char 배열을 출력하기 위한 함수 중복
    for(int i=0; i<n; i++)
        cout << (int)array[i] << '₩t'; // array[i]를 int 타입으로 변환하여 정수 출력
    cout << endl;
}

int main() {
    int x[] = {1,2,3,4,5};
    double d[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    print(x, 5);
    print(d, 5);

    char c[5] = {1,2,3,4,5};
    print(c, 5);
}
```

템플릿 함수와 중복된 print() 함수

중복된 print() 함수가 우선 바인딩

템플릿 print() 함수로부터 구체화

```
1       2       3       4       5
1.1     2.2     3.3     4.4     5.5
1       2       3       4       5
```

주목