03. 개선된 함수 기능

3.1 인라인 함수

3.2 디폴트 인자

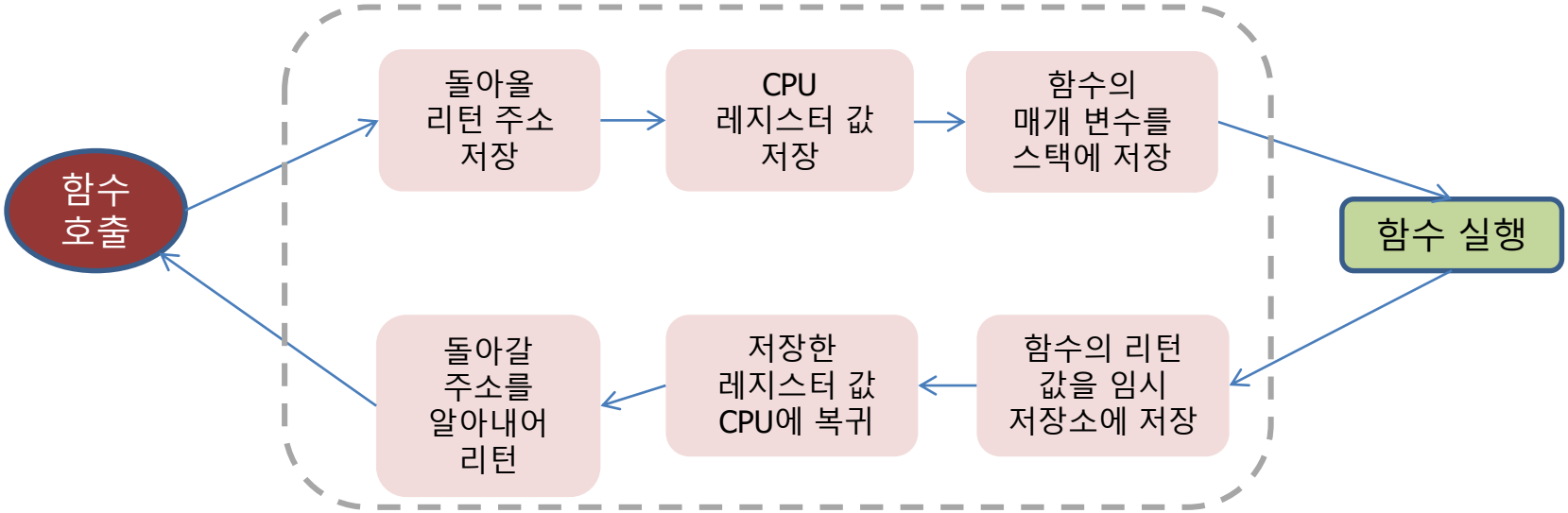
3.3 함수 중복(오버로딩)

3.4 함수 템플릿

3.1 인라인 함수

함수 호출에 따른 시간 오버헤드

- 인라인 함수란?
  - 함수 호출 시 발생하는 오버헤드를 줄이기 위해서 함수를 호출하는 대신 함수가 호출되는 곳마다 함수의 코드를 복사하여 넣어주는 특별한 함수



함수 호출에 따른 시간 오버헤드

작은 크기의 함수를 호출하면, 함수 실행 시간에 비해, 호출을 위해 소요되는 부가적인 시간 오버헤드가 상대적으로 크다.

함수 호출에 따른 오버헤드가 심각한 사례

- 실제 계산 시간 보다 함수 호출에 따른 오버헤드가 더 큰 사례

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

int odd(int x) {
   return (x%2);
}

int main() {
   int sum = 0;

   // 1에서 10000까지의 홀수의 합 계산
   for(int i=1; i<=10000; i++) {
      if(odd(i))
         sum += i;
   }
   cout << sum;
}
```

10000번의 함수 호출.
호출에 따른 엄청난
오버헤드 시간이 소모됨.

odd() 함수의 코드를 계산하는 시간보다 odd() 함수 호출에 따른 오버헤드가 더 크며, 루프를 돌게 되면 오버헤드는 더욱 가중됨.

```
25000000
```

인라인 함수

인라인 함수

- 함수의 선언이나 정의에 **inline** 키워드를 지정하여 선언된 함수

인라인 함수에 대한 처리

- 인라인 함수를 호출하는 곳에 인라인 함수 코드를 확장 삽입
  - 매크로와 유사
  - 코드 확장 후 인라인 함수는 사라짐
- 인라인 함수 호출
  - 함수 호출에 따른 오버헤드 존재하지 않음
  - 프로그램의 실행 속도 개선
- 컴파일러에 의해 이루어짐

인라인 함수의 목적

- C++ 프로그램의 실행 속도 향상
  - 자주 호출되는 짧은 코드의 함수 호출에 대한 시간 소모를 줄임
  - C++에는 짧은 코드의 멤버 함수가 많기 때문

인라인 함수의 사용 예

- 컴파일러에 의한 코드 대치

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

inline int odd(int x) {
   return (x%2);
}

int main() {
    int sum = 0;
    for(int i=1; i<=10000; i++) {
        if(odd(i)) sum += i;
    }
    cout << sum;
}
```

컴파일러에 의해 inline 함수의 코드 확장 삽입

컴파일러는 inline 처리 후, 확장된 C++ 소스 파일을 컴파일 한다.

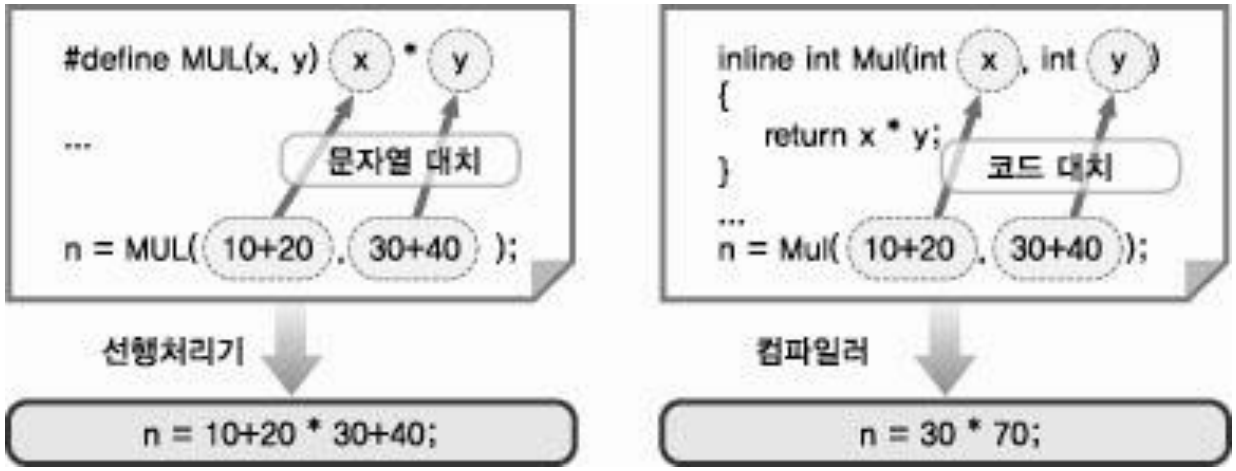
```
#include <iostream>
using namespace std;

int main() {
    int sum = 0;

    for(int i=1; i<=10000; i++) {
        if((i%2)) sum += i;
    }
    cout << sum;
}
```

인라인 제약 사항
- inline은 컴파일러에게 주는 요구 메시지
- 컴파일러가 판단하여 inline 요구를 수용할 지 결정
- recursion, 긴 함수, static, 반복문, goto 문 등을 가진 함수는 수용하지 않음

인라인 함수 vs 매크로 함수

- 매크로 함수와 인라인 함수의 차이점
  - 매크로 함수 ➔ 선행처리기에 의한 문자열 대치 방식
  - 인라인 함수 ➔ 컴파일러에 의한 코드 대치 방식

Macro.cpp



- 매크로 함수의 문제점
  - 연산자 우선 순위 문제
  - 인자의 형 검사를 하지 않음

➡ **인라인 함수를 사용하는 것이 좋다**

인라인 함수 장단점

- 장점
  - 프로그램의 실행 시간이 빨라진다.

- 단점
  - 인라인 함수 코드의 삽입으로 컴파일 된 전체 코드 크기 증가
    - 통계적으로 최대 30% 증가
    - 짧은 코드의 함수를 인라인으로 선언하는 것이 좋음

3.2 디폴트 인자

디폴트 인자(default parameter)

- 디폴트 인자란?
  - 인자에 값이 넘어오지 않는 경우, 디폴트 값을 받도록 선언된 인자
    - '**인자 = 디폴트값**' 형태로 선언

- 디폴트 인자를 지정할 때는 **함수의 선언부**에 지정
  - 디폴트 인자를 지정할 때는 인자형. 인자 이름 다음에 = 과 함께 디폴트 값을 지정한다.
  - 함수를 정의할 때는 기존 함수를 정의하는 방법대로 함.

디폴트 인자

```
void default_sample(char c, int i,  double d = 0.5 );  // 함수의 선언부
void main() {
    default_sample ('X', 10);
    default_sample ('Y', 30, 2.0);
}
```

디폴트 인자에 관한 제약 조건

- 디폴트 인자 지정 순서
  - 함수의 가장 오른쪽 인자부터 지정해야 한다.

```
void foo1(char c, int i, double d);
void foo2(char c, int i, double d = 0.5);
void foo3(char c, int i = 10, double d = 0.5);
void foo4(char c = 'A', int i = 10, double d = 0.5);
```

- 함수 인자의 생략
  - 함수의 가장 오른쪽 인자부터 생략해야 한다.

```
void foo(char c = 'A', int i = 10, double d = 0.5);
foo('A', 20);                    // 세 번째 인자만 생략함
foo('B');                        // 두 번째, 세 번째 인자만 생략함
foo();                           // 인자 모두를 생략함
```

디폴트 인자 사례

void g(int a, **int b=0, int c=0, int d=0**);

디폴트 매개 변수를 가진 함수

void g(int a, int b= 0, int c= 0, int d= 0);

g(10);	→	g(10, _ , _ , _);	→	g(10, 0, 0, 0);
g(10, 5);	→	g(10, 5 , _ , _);	→	g(10, 5, 0, 0);
g(10, 5, 20);	→	g(10, 5, 20, _);	→	g(10, 5, 20, 0);
g(10, 5, 20, 30);	→	g(10, 5, 20, 30);	→	g(10, 5, 20, 30);

컴파일러에 의해 변환되는 과정

디폴트 매개 변수를 가진 함수 선언 및 호출

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

enum INT_TYPE {DECIMAL, OCTAL, HEXADECIMAL};
void PrintArray(const int arr[], int size = 5, INT_TYPE type = DECIMAL);

int main() {
    int arr1[] = {10, 20, 30, 40, 50};
    int arr2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    PrintArray(arr1);
    PrintArray(arr1, 5, HEXADECIMAL);
    PrintArray(arr2);
    PrintArray(arr2, 10);

    return 0;
}

void PrintArray(const int arr[], int size, INT_TYPE type) {
    cout.setf(ios::showbase);
    for(int i = 0 ; i < size ; i++)  {
        switch( type )  {
          case DECIMAL:         cout << dec;    break;
          case OCTAL:           cout << oct;    break;
          case HEXADECIMAL:     cout << hex;    break;
        }
        cout.width(5);
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

3.3 함수 중복(오버로딩)

함수의 중복

- 함수 중복 = 함수 오버로딩(function overloading)
  - 이름은 같지만 인자의 개수나 데이터 형이 다른 함수를 여러 번 정의할 수 있는 기능
    - 다형성
    - C 언어에서는 불가능
  - 함수 중복 기능을 사용하면 같은 이름을 갖는 함수를 내용물이 다르도록 여러 번 정의할 수 있다.
  - 각각의 함수는 인자의 시그니처(인자의 개수나 인자의 데이터 형)가 반드시 달라야 함.
  - 함수 호출 시 전달된 인자의 데이터 형으로 컴파일러가 어떤 함수를 호출할 지를 결정

- 함수 중복 성공 조건
  - 중복된 함수들의 이름 동일
  - 중복된 함수들의 매개 변수 타입이 다르거나 개수가 달라야 함
  - 리턴 타입은 함수 중복과 무관

함수 중복 성공 사례

```
int sum(int a, int b, int c) {
   return a + b + c;
}

double sum(double a, double b) {
   return a + b;
}

int sum(int a, int b) {
   return a + b;
}
```

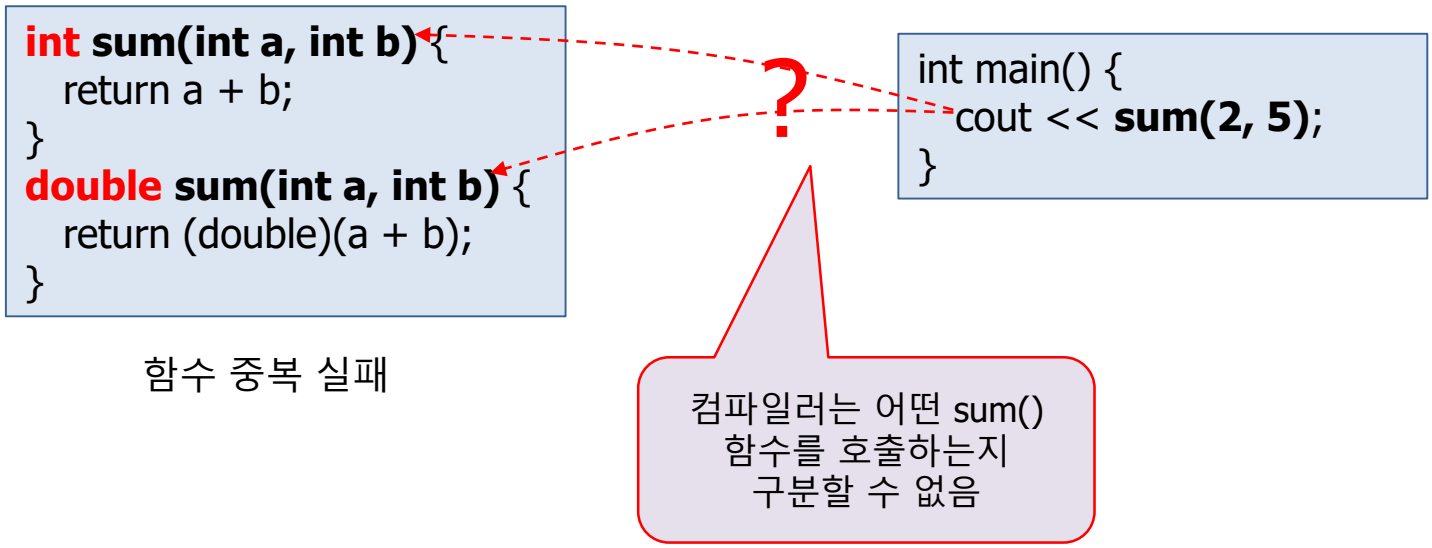
성공적으로 중복된 sum() 함수들

```
int main(){
   cout << sum(2, 5, 33);

   cout << sum(12.5, 33.6);

   cout << sum(2, 6);
}
```

중복된 sum() 함수 호출.
컴파일러가 구분

함수 중복 실패 사례

- 리턴 타입이 다르다고 함수 중복이 성공하지 않는다.



함수 중복 실패

함수 중복의 편리함

- 동일한 이름을 사용하면 함수 이름을 구분하여 기억할 필요 없고, 함수 호출을 잘못하는 실수를 줄일 수 있음

```
void msg1() {
   cout << "Hello";
}
void msg2(string name) {
   cout << "Hello, " << name;
}
void msg3(int id, string name) {
   cout << "Hello, " << id << " " <<
name;
}
```

(a) 함수 중복하지 않는 경우

```
void msg() {
   cout << "Hello";
}
void msg(string name) {
   cout << "Hello, " << name;
}
void msg(int id, string name) {
   cout << "Hello, " << id << " " << name;
}
```

(b) 함수 중복한 경우

함수 중복하면 함수 호출의 편리함.오류 가능성 줄임

big() 함수

- 큰 수를 리턴하는 두 개의 big 함수

```
int big(int a, int b);        // a와 b 중 큰 수 리턴
int big(int a[], int size);   // 배열 a[]에서 가장 큰 수 리턴
```

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

int big(int a, int b) { // a와 b 중 큰 수 리턴
   if(a>b) return a;
   else return b;
}

int big(int a[], int size) { // 배열 a[]에서 가장 큰 수 리턴
   int res = a[0];
   for(int i=1; i<size; i++)
      if(res < a[i]) res = a[i];
   return res;
}

int main() {
   int array[5] = {1, 9, -2, 8, 6};
   cout << big(2,3) << endl;
   cout << big(array, 5) << endl;
}
```

```
3
9
```

함수 중복 시 주의사항

- 인자의 이름만 다른 경우 오버로딩 할 수 없다.

```
int foo1(int a, int b);
int foo1(int x, int y);
foo1(10, 20);
```

- 함수의 리턴형만 다른 경우 오버로딩 할 수 없다.

```
int foo2(char c, int num);
void foo2(char c, int num);
foo2('A', 20);
```

- 데이터 형과 해당 형의 레퍼런스 형으로 오버로딩된 경우 오버로딩할 수 없다.

```
int foo3(int a, int b);
int foo3(int &a, int &b);
int x = 10, y = 20;
foo3(x, y);
```

함수 중복 시 주의사항

- typedef로 정의된 데이터 형에 대해 오버로딩된 경우 오버로딩 할 수 없다.

```
typedef unsigned int UINT;
void foo4(UINT a, UINT b);
void foo4(unsigned int a, unsigned int b);
foo4(10, 20);
```

- 디폴트 인자에 의해서 인자의 개수가 같은 경우 오버로딩 할 수 없다.

```
void foo5(int a);
void foo5(int a, int b = 0);
foo5(10);
```

디폴트 인자 vs 함수 중복

- 디폴트 인자의 사용
  - 두 함수의 처리 과정이 비슷하고 한 함수가 다른 함수의 특별한 경우로 간주되는 경우

```
int GetSum(int x, int y, int z = 0)  {          // 디폴트 인자 지정
    return x + y + z;
}

int main()   {
    GetSum(10, 20);                             // GetSum(10, 20, 0); 호출
    GetSum(10, 20, 30);
}
```

디폴트 인자 vs 함수 중복

- 함수 오버로딩의 사용
  - 두 함수의 구체적인 처리 과정이 다르지만 같은 함수 이름으로 표현될 수 있다는 공통점만 갖는 경우

```
int GetSum(int x, int y) {
    return x + y;
}

int GetSum(const int arr[], int size)     {
    int sum = 0;
    for(int i = 0 ; i < size ; i++)
        sum += arr[i];
    return sum;
}
```

디폴트 인자를 사용한 함수 중복 간소화

```
void fillLine() { // 25 개의 '*' 문자를 한 라인에 출력
   for(int i=0; i<25; i++)   cout << '*';
   cout << endl;
}
void fillLine(int n, char c) { // n개의 c 문자를 한 라인에 출력
   for(int i=0; i<n; i++)   cout << c;
   cout << endl;
}
```

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

void fillLine(int n=25, char c='*') { // n개의 c 문자를 한 라인에
출력
   for(int i=0; i<n; i++) cout << c;
   cout << endl;
}

int main() {
   fillLine(); // 25개의 '*'를 한 라인에 출력
   fillLine(10, '%'); // 10개의 '%'를 한 라인에 출력
}
```

3.4 함수 템플릿

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

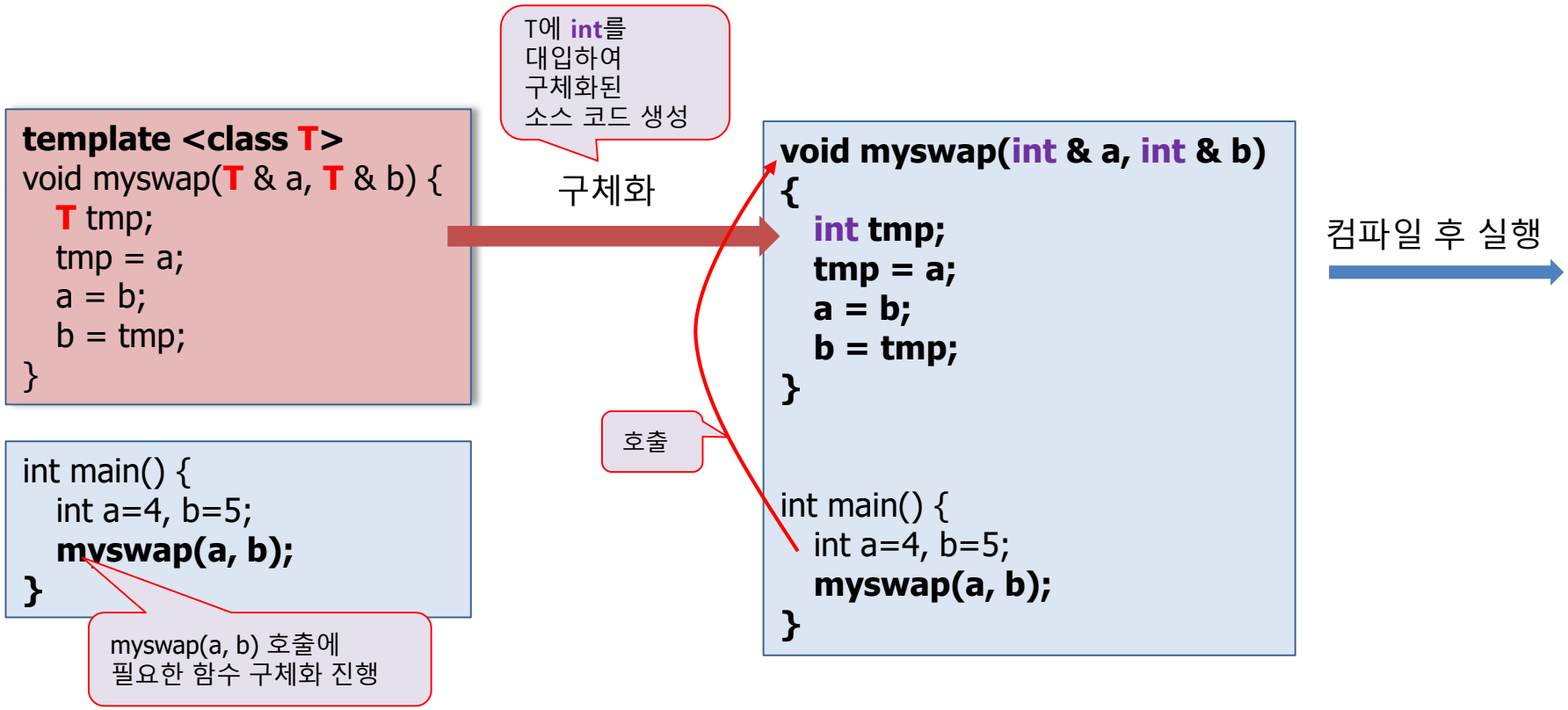
제네릭 타입 T 선언

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

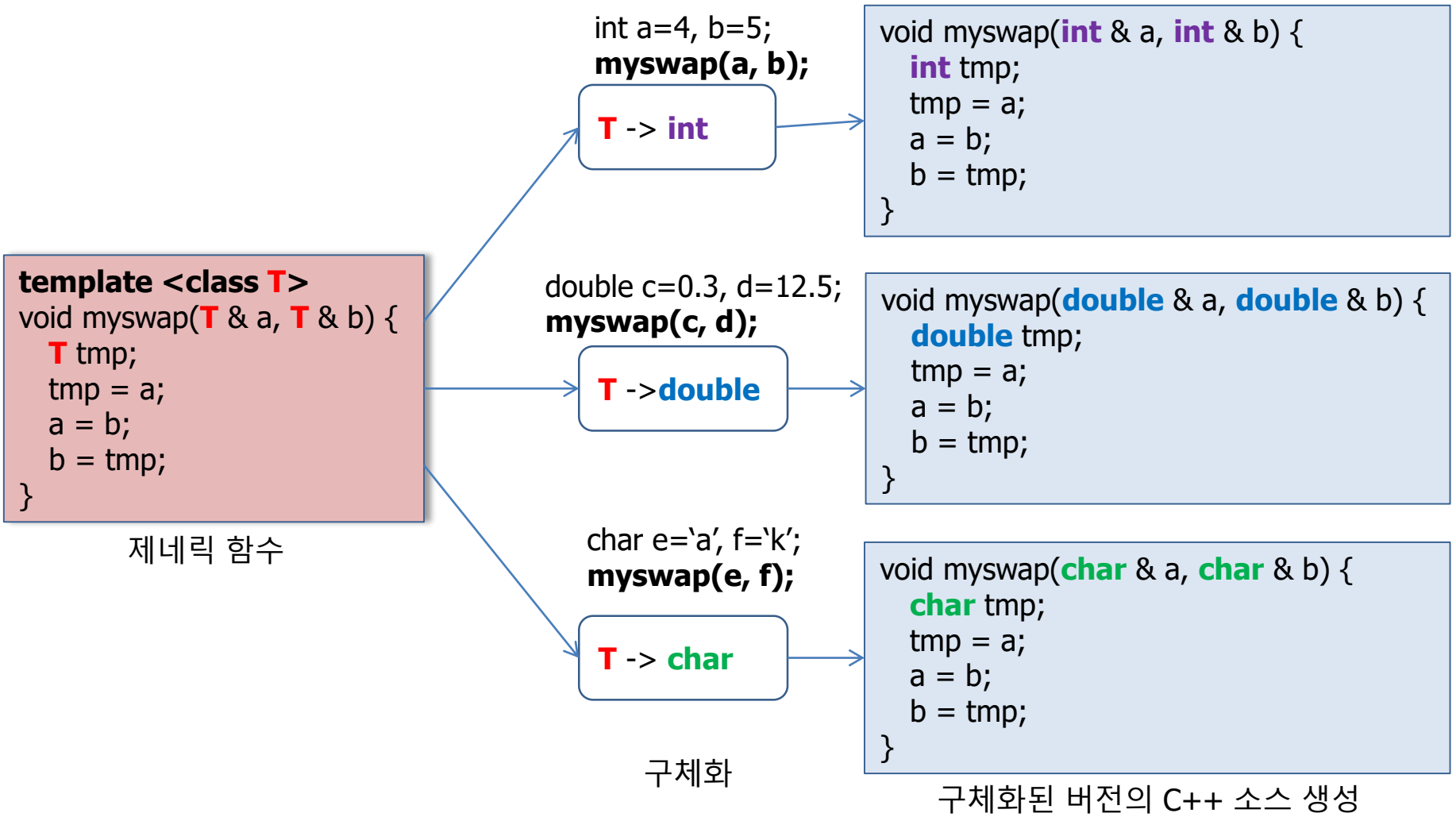
템플릿을 이용한 제네릭 함수

템플릿으로부터의 구체화

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



제네릭 함수로부터 구체화된 함수 생성 사례



템플릿으로부터 구체화 예시

- 암시적 인스턴스화

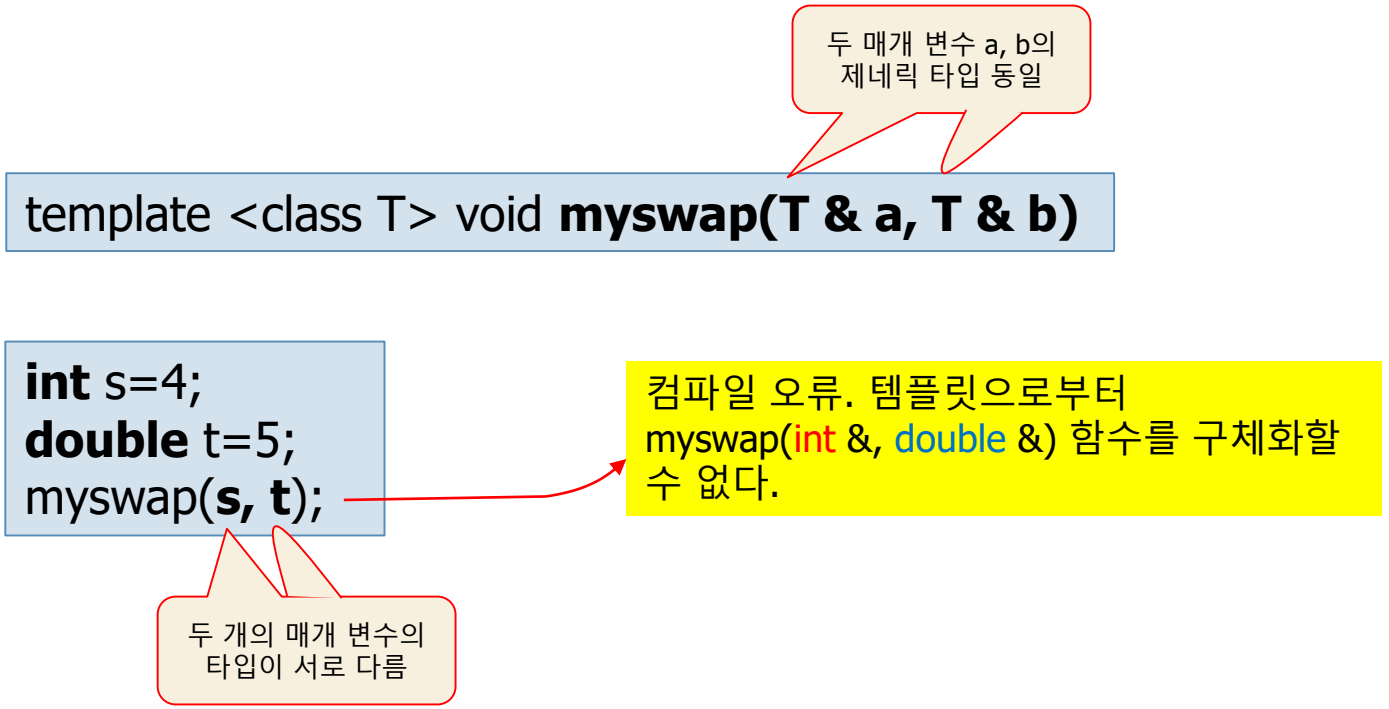
```
cout << GetMax(10, 20) << endl;                 // T = int
char ch = GetMax('A', 'B');                     // T = char
cout << GetMax(3.14, 10.5) << endl;             // T = double
cout << GetMax(5, 10.5) << endl;                // 컴파일 에러
```

- 명시적 인스턴스화

```
cout << GetMax<int>(5, 10.5) << endl;           // T= int
cout << GetMax<double>(5, 10.5) << endl;        // T= double
```

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

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

다음 수업

- 클래스와 객체의 기본
  - 1_ 클래스와 객체의 기본개념
  - 2_ string 클래스와 vector 클래스
  - 3_ C++에서의 클래스 정의 및 사용