중복 함수들의 약점


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

```
double Add(double a, double b)
{
    return a+b;
}
```

중복 함수들

두 함수는 매개변수만 다르고 코드는 동일

함수 중복작성은 프로그램의 길이도 늘어나고 작성시 실수도 늘어나며, 알고리즘을 수정하게 되면 중복된 함수 모두를 변경해야 하는 번거로움이 있다.

일반화와 템플릿

- 제네릭(generic) 또는 일반화
  - 함수나 클래스를 일반화시키고, 매개 변수 타입을 지정하여 틀에서 찍어 내듯이 함수나 클래스 코드를 생산하는 기법
  - 자료형에 구애 받지 않고 하나의 로직으로 처리하는 함수는 일반화(generic) 함수 라 한다.
- 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(template)에 대한 이해

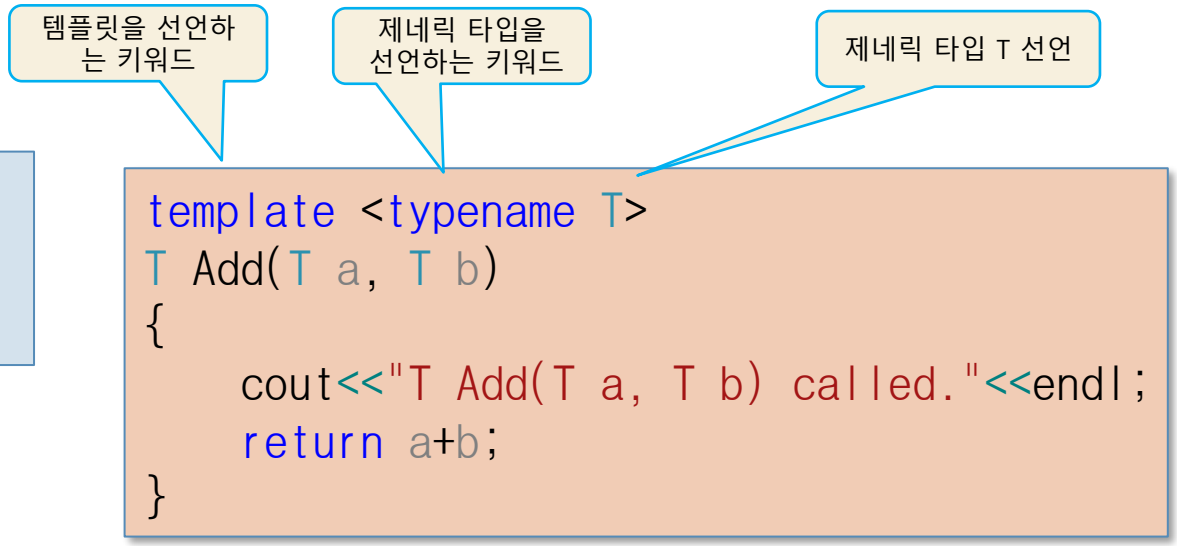
- **함수 템플릿** :컴파일러는 인자의 형을 검사하여 함수를 생성한다.
- 일반화(Generic) : 형은 다양하지만 알고리즘은 동일한 함수를 만들 때 유용하다.
- **template <typename T> : T는 자료형을 결정 짓지 않겠다. 자료형이 결정되는 시점은 인자가 전달되는 순간이다.**

일반화와 템플릿

템플릿 선언

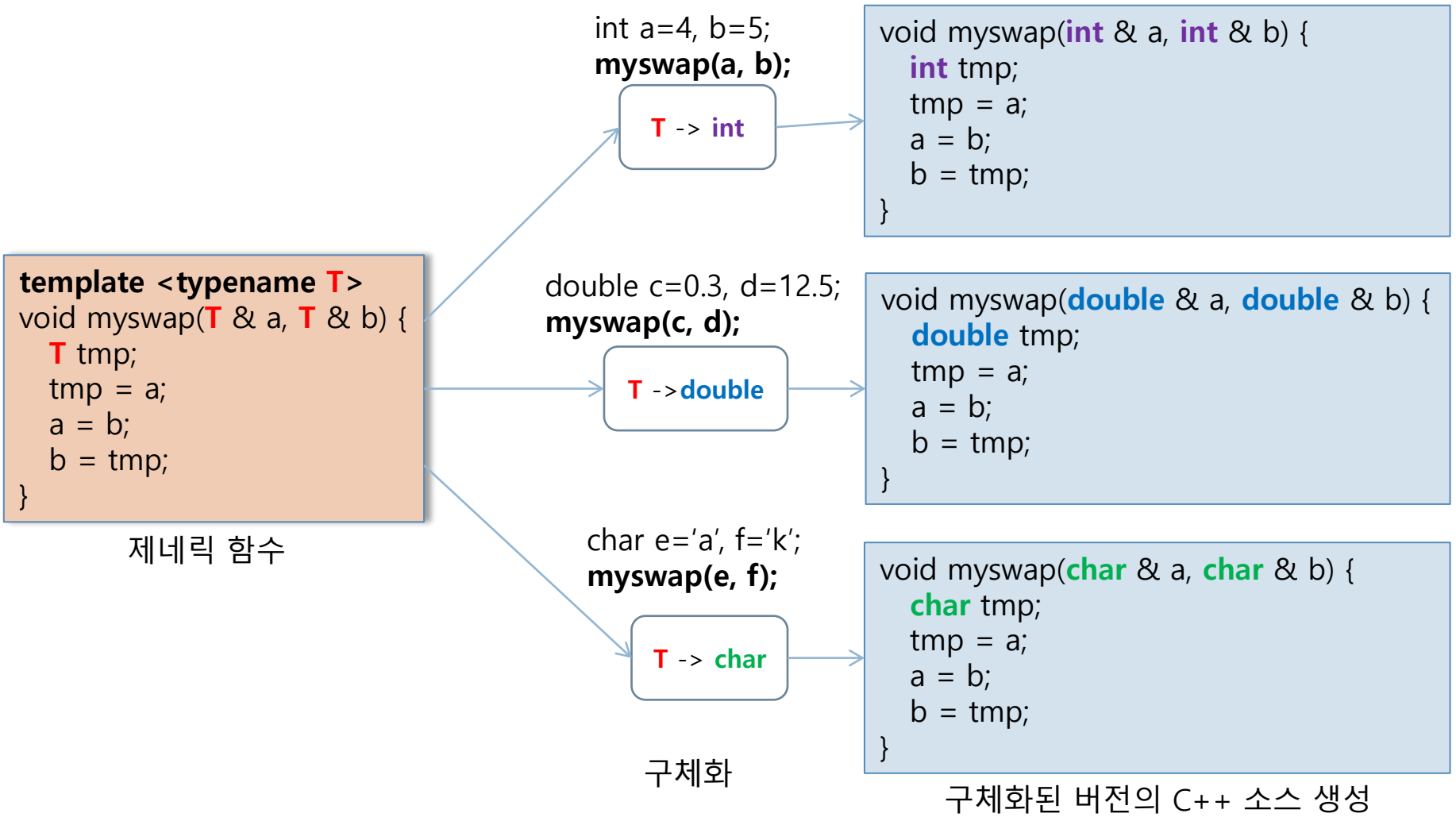
template <class T> 또는
template <typename T>

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



템플릿을 이용한 제네릭 함수 Add

제네릭 함수로부터 구체화된 함수 생성 사례



구체화 오류

- 제네릭 타입에 구체적인 타입 지정 시 주의

두 매개 변수 a, b의 제네릭 타입 동일

```
template < typename T>
void myswap(T & a, T & b)
```

```
int s=4;
double t=5;
myswap(s, t);
```

컴파일 오류. 템플릿으로부터 myswap(int &, double &) 함수를 구체화할 수 없다.

두 개의 매개 변수의 타입이 서로 다름

다음과 같이 여러 개의 제너릭 타입을 구분할 수 있다.

```
template < typename T1, typename T2 >
void myswap(T1 & a, T2 & b)
```

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

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!

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

1 2 3 4 6

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

문제풀이


```
/*
다음의 실행결과를 만드는 템플릿을 작성하시오

4
8
13
*/

int main()
{
    int i = 10;
    double d = 7.7;
    const char* str = "Good morning!";

    cout << SizeOf(i) << endl;
    cout << SizeOf(d) << endl;
    cout << SizeOf(str) << endl;

    return 0;
    }
```

문제풀이

- /*
- 다음과 같이 실행하려고 한다.

- 실행결과
- 30
- 3.3
- Kingdom
- Multi Campus
- */

```
int main()
{
    char s1[20] = "King", s2[20] = "dom";

    cout << Add(10, 20) << endl;
    cout << Add(1.1, 2.2) << endl;
    cout << Add(1.1, 2.2) << endl;

    cout << Add(s1, s2) << endl;              //오류
    cout << Add("Multi ", "Campus") << endl; //오류


    return 0;
}
```

문제풀이


```
/*
다음의 실행결과를 만드는 템플릿을 작성하시오
실행결과
60
102.2
15
*/
int main()
{
	int arr1[]={10, 20, 30};
	cout<<SumArray(arr1, sizeof(arr1)/sizeof(int))<<endl;

	  double arr2[]={10.3, 20.4, 30.5};
	  cout<<SumArray<double>(arr2,
		  sizeof(arr2)/sizeof(double))<<endl;

	  int arr3[] = { 1,2,3,4,5 };
	  cout << SumArray(arr3) << endl;
   }
```

제네릭 클래스 만들기

template을 이용하면 제너릭 클래스를 만들 수 있다. 이는 타입이 다른 여러 자료형 클래스를 생성한다.

제네릭 클래스 선언

```
template <typename T>
class Data
{
private:
        T data;
public:
        Data(T d)
        {
                data = d;
        }
        void SetData(T d)
        {
                data = d;
        }
        T GetData()
        {
                return data;
        }
};
```

클래스 구체화 및 객체 활용

```
Data<int> d1(0);
d1.SetData(10);
Data<char> d2('A');
```

클래스 템플릿

```
template <typename T>
class Data
{
    T data;
public:
    Data(T d){ data=d; }
    void SetData(T d){
        data=d;
    }
    T GetData(){
        return data;
    }
};
```

선언/정의 분리

```
template <typename T>
class Data
{
    T data;
public:
    Data(T d);
    void SetData(T d);
    T GetData();
};

template <typename T>
Data<T>::Data(T d){
    data=d;
}
template <typename T>
void Data<T>::SetData(T d){
    data=d;
}
template <typename T>
T Data<T>::GetData(){
    return data;
}
```

템플릿 사용 시 유의할 점

- 템플릿은 컴파일 시간에 코드를 만들어낸다.
  - 그렇기 때문에 템플릿의 사용으로 인한 속도 저하는 생기지 않는다.
  - 하지만, 컴파일 시간이 오래 걸리는 단점이 있다. (하지만 크게 문제가 될 정도는 아님)
- 템플릿 함수의 정의는 헤더 파일에 놓여야 한다.
  - 컴파일 시간에 클래스나 함수를 만들어내기 위해서는 헤더 파일에 위치할 필요가 있다. (컴파일 시간에는 다른 구현 파일의 내용을 확인할 수 없다.)

문제

```
int main()
{
    //클래스 템플릿은 제너릭  T에 적용할 타입을 지정해야 한다.
    Point<int> pos1(10, 20);
    Point<int> pos2(100, 200);

    pos1.ShowPosition();
    pos2.ShowPosition();
    cout << endl;

    //SwapData 호출
    SwapData(pos1, pos2);
    pos1.ShowPosition();
    pos2.ShowPosition();
    cout << endl;
    cout << "======================" << endl;

    Point<double> pos3(1.1, 2.2);
    Point<double> pos4(5.1, 6.2);
    SwapData(pos3, pos4);
    pos3.ShowPosition();
    pos4.ShowPosition();
```

실행결과

```
[10, 20]
[100, 200]

[100, 200]
[10, 20]

==========
====
[5.1, 6.2]
[1.1, 2.2]
```

문제

```
class Point
{
private:
        int x, y;
public:
        Point(int x = 0, int y = 0) : x(x), y(y)
        { }
        void ShowPosition() const
        {
                cout << '[' << x << ", " << y << ']' << endl;
        }
};
```

```
int main()
{
        Point <int>p1(1, 2);
        Point <int>p2(1, 2);

        Point <int>p3 = Add(p1 , p2);
        p3.ShowPosition();
        cout << "==================

        Point<double> p4(1.1, 2.2);
        Point<double> p5(1.1, 2.2);

        Point <double>p6 = Add(p4, p5);
        p6.ShowPosition();
}
```

실행결과

```
[2, 4]
========
[2.2, 4.4]
```

두 제너릭 타입을 가진 클래스

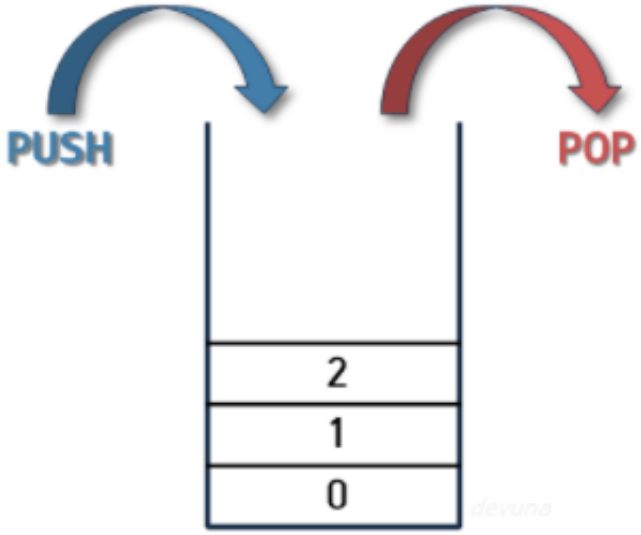
템플릿을 이용하여 2개 이상의 제너릭 타입을 가진 제너릭 클래스를 만들 수 있다.

template <class T1, class T2, class T3>

```
template <class T1, class T2> // 두 개의 제네릭 타입 선언
class GClass
{
      T1 data1;
      T2 data2;
   public:
      GClass();
      void set(T1 a, T2 b);
      void get(T1 &a, T2 &b);
};
```

stack

스택 구조는 마지막에 저장한 데이터를 먼저 사용하는 구조이다.



LIFO(Last In First Out) 방식

스택(stack)이란 **쌓아 올린다는 것**을 의미한다.

따라서 스택 자료구조라는 것은 책을 쌓는 것처럼 **차곡차곡 쌓아 올린 형태의 자료구조**를 말한다.

stack


```
class Stack
{
private:
   int topIdx;   // 마지막 입력된 위치의
인덱스.
   char* stackPtr;  // 스택 포인터.
   int size;
public:
Stack(int len=10);
~Stack();
void Push(const char& pushValue);
char Pop();
};
```

📌 스택의 특징

스택은 위의 사진처럼 **같은 구조와 크기의 자료**를 **정해진 방향으로만** 쌓을수 있고,

top으로 정한 곳을 통해서만 접근할 수 있다.

top에는 가장 위에 있는 자료는 가장 최근에 들어온 자료를 가리키고 있으며,

삽입되는 새 자료는 top이 가리키는 자료의 위에 쌓이게 된다.

스택에서 자료를 삭제할 때도 top을 통해서만 가능하다.

스택에서 top을 통해 **삽입하는 연산**을 'push' , top을 통한 **삭제하는 연산**을 'pop'이라고

다양한 자료형을 가지는 stack


```
template <typename T>
class Stack
{
    private:
        int topIdx;   // 마지막 입력된 위치의 인덱스.
       T* stackPtr;   // 스택 포인터.
       int size;
    public:
        Stack(int s=10);
        ~Stack();
        void Push(const T& pushValue);     //Push(const char*&
        pushValue)
        T Pop();

        int Get_topIdx()
        {
                return topIdx;
        }
};
```

stack 문제풀이


```
template <class T>
class MyStack
{
	int top;	// top of stack
	T data[100]; // T 타입의 배열. 스택의 크기는 100
	public:
		MyStack();
		void push(T element); //T 타입 원소 element를 data [] 배열에 삽입
		T pop();		//스택의 탑에 있는 데이터를 data[] 배열에서 리턴

		int Get_topIdx()
		{
			return top;
		}
};

class Point
{
	int x, y;
	public:
	Point(int x = 0, int y = 0) { this->x = x; this->y = y; }
	void show() { cout << '(' << x << ',' << y << ')' << endl; }
};
```

stack 문제풀이

다음과 같이 출력하는 스택구조를 완성하시오

실행결과

```
====== MyStack<int *> ipStack =========
100 200 300 400
0 10 20
====== MyStack<Point> pointStack ==========
(1,4)
(1,3)
(1,2)
(1,1)
======= MyStack<Point*> pStack =========
(10,30)
(10,20)
(10,10)
====== MyStack<string> stringStack =========
C# ,python,C ,java,C++,
```

변환생성자

변환 생성자 : 생성자 중 인자가 하나인 생성자를 변환생성자 라고도 함.

EX) 객체=정수값;

객체는 정수를 대입 받을 수 없지만 생성자 중 정수를 인자로 갖는 생성자가 있으면 생성자(**변환생성**자) 호출 된다.

```
class Type  // => 사용자 정의 클래스
{
        int a;
public:
    Type(int _tmp)  //매개변수가 한 개인 생성자를 변환 생성자라 함.
    {
        cout<<"Type(int _tmp) called." <<endl;
        a=_tmp;
    }
}
```

형변환 연산자

형변환 연산자(메서드): 객체는 정수 값을 바로 대입 받을 수 없다.

EX) 정수형변수 =객체; (int num=객체명)

그러나 객체가 정수형 멤버를 포함하며 이를, int형으로 형변환해야 하는 상황에서 **int() 메서드**가 호출된다. 이 메서드를 **형변환 연산자라** 한다

```
operator int()
{
    cout << "operator int() called." << endl;
    return a;
}
```

문제풀이,

다음과 같이 다양한 자료형을 형변환을 사용하여 실행결과를 출력한다.

```
Type<int> data1(100);
data1 = 200;        //생성자 call
int num = data1;    //operator int() 호출
cout << num<<endl;
cout<<"==================="<<endl;

Type<char> data2('A');
char ch = data2;     //operator char() 호출
cout << ch <<endl;
cout<<"==================="<<endl;

Type<double> data3(3.5);
double dnum = data3;
cout << dnum<<endl;
cout<<"==================="<<endl;

Type<string> data4("multi campus");
string str = data4;
cout << str<<endl;
```

실행결과

```
Type(T _tmp) called.
Type(T _tmp) called.
operator T() called.
200
===================
Type(T _tmp) called.
operator T() called.
A
===================
Type(T _tmp) called.
operator T() called.
3.5
===================
Type(T _tmp) called.
operator T() called.
multi campus
```

문제풀이


```
class CMyString
{
  char* str;
  int len;
public:
  CMyString(…)  //생성자
  {

  }
  //소멸자
  //형변환 연산자
  }
```

실행결과

```
CMyString(const char * _tmp) called.
operator char*() const.
TestFunc(const CMyString& strParam) : Hello
==============
CMyString(const char * _tmp) called.
operator char*() const.
TestFunc(const CMyString& strParam) : World
~CMyString() called.
~CMyString() called.
```

문제풀이


```
void TestFunc(…)
{
    cout <<"TestFunc(const CMyString& strParam) : "<< strParam<<endl;
}

int main()
{
    CMyString strData("Hello");

    TestFunc(strData);
    cout<<"==============="<<endl;

   TestFunc(CMyString("World"));
}
```

실행결과

```
CMyString(const char * _tmp) called.
operator char*() const.
TestFunc(const CMyString& strParam) : Hello
==============
CMyString(const char * _tmp) called.
operator char*() const.
TestFunc(const CMyString& strParam) : World
~CMyString() called.
~CMyString() called.
```

R_value 참조

R_value는 단순 대입 연산자의 오른쪽 항을 말한다. 변수나 상수(100, 1.5, 'A') 같은 값이 r_value 이다.

상수이므로 변수가 아닌 대상에 참조를 선언하는 것은 허용되지 않았으나, 최근 r_value에 대한 참조자를 새롭게 제공하여 상수를 참조할 수 있게 되었다.

r_value 참조자는 기존 참조자와 달리 &가 두 번 붙는다. int 형에 대한 참조는 int& 가 되지만, int 자료형에 대한 r_value 참조형식은 int && 가 된다.

```
int x =100;
int& x1 = x;   //참조변수

int&& y = 3 + 4; //r_value 참조변수
```

배열 index 오버로딩

- 객체를 배열처럼 , **배열 인덱스 연산자 오버로딩**
- C,C++의 기본 배열은 경계검사를 하지 않는 약점을 가지고 있다. C++에선 이 취약점을 **인덱스 연산자 오버로딩을** 통해 극복할 수 있다.
- int operator[](int idx){...} 와 같이 오버로딩 되었다면,arrObject[2] 는 arrObject.operator[](2)로 해석된다.
- 멤버함수로서 operator[]를 호출한 것이다. 이 오버로딩 함수안에서 경계검사를 진행하여 안전성을 확보하게 된다.

객체를 배열처럼

하나의 자료형으로 배열을 만들는 클래스, 객체를 배열처럼 사용 접근한다.

```
int& Arr::operator[](int i)
{
        if(i<0 || i>=SIZE)
        {
           cout<<"배열 첨자오류!!!"<<endl;
           exit(1);
        }

        idx = i;
        return nArr[i];   //*(nArr+i)
}
```

객체를 배열처럼

객체 동적할당 받아 객체를 배열처럼 접근한다.

```
private:
int* ArrPtr;
int idx;
public:
    AutoArray(int* ptr)
    {

    }
    ~AutoArray()
    {

    }
…
}
```

```
int main()
{
 AutoArray arr(new int[100]);

// 객체 배열 사용
    arr[0]=10;
    arr[1]=20;
    arr[2]=1000;
    arr[3]=2000;
    //arr[100]=2000; //out of range

    int idx = arr.GetIdx();
    for(int i=0;i<=idx;i++)
        cout<<arr[i]<<endl; …
}
```

다양한 자료형으로 배열을 만들는 템플릿 클래스

문제] class AutoArray 클래스를 수정하여 다음의 결과를 출력하시오.

```
int main()
{
//객체배열 생성
    AutoArray<int> arrInt(new int[A_SIZE]);
    AutoArray<char> arrChar(new char[A_SIZE]);
    AutoArray<double> arrDouble(new double[A_SIZE]);
    AutoArray<string> arrStr(new string[A_SIZE]);

   //객체 배열 사용
    arrInt[0]=1;
    arrChar[0]='A';
    arrDouble[0]=5.1;
    arrStr[0]="campus";
    //arrDouble[12]=9.1; //out of range

    // 객체배열 출력
   cout<<"₩n"<<arrInt[0]<<", "<<arrChar[0]<<", "<<arrDouble[0]
            << ", "<< arrStr[0] << endl;
   cout <<endl;
}
```

```
실행결과
T& operator[] (int index) called => 0
T& operator[] (int index) called => 0
…
T& operator[] (int index) called => 0
T& operator[] (int index) called => 0
1, A, 5.1, campus
```