게임프로그래밍

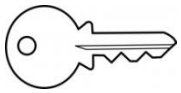
템플릿

박종승

Dept. of CSE, Incheon Nat. Univ.
jong@inu.ac.kr
http://ecl.inu.ac.kr

목차

- 이름공간
- 함수템플릿
- 클래스템플릿
- const
- 형변환



이름공간

- 이름공간(name space) 사용의 필요성
  - 많은 library의 사용과 코드의 대형화
    → 변수나 함수의 이름의 충돌 문제가 발생
- 이름공간의 생성
  - namespace 키워드를 사용
    - 이름공간 내에는 상수, 변수, 함수, 클래스 등 모두 포함시킬 수 있음

```
namespace first
{
    int x = 5;
}
```

이름공간'

- 이름공간의 사용
  - C++ 입출력관련 library에는 std라는 이름공간이 있음
    - 표준 입출력과 관련된 기능들이 구현되어 있음
    - cin, cout, endl 등의 식별자(identifier)들이 정의되어 있음

```
std::cout << "x=" << first::x << std::endl;
```

  - 이름공간을 일일이 명시하는 것이 귀찮다면
    - using 선언(declaration)을 사용

```
using std::cout;
using std::endl;
using first::x;
cout << "Hello" << x << endl;
```

    - using 지시어(directive)를 사용하면

```
using namespace std;
using namespace first;
cout << "Hello" << x << endl;
```

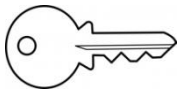
이름공간”

- 식별자의 범위
  - 이름공간을 명시하지 않고 정의된 식별자

```
int i = 1;
namespace first
{
    int i = 2;
    namespace second
    {
        int j = i+2;
    }
}
cout << i << ", " << first::i << ", " << first::second::j << endl;
```

    - 이름공간 second 내에서 언급된 i
      - 먼저, second에서 i를 찾음
      - 없으면, first에서 찾음
      - 없으면, 전역 이름공간에서 찾음
  - 이름공간이 전역공간임을 명확히 명시적으로 표시
    - 식별자 앞에 '::'를 붙임

```
cout << ::i << ", " << first::i << ", " << first::second::j << endl;
```



함수템플릿

01.TemplateMax

- 템플릿(template)
  - 각 데이터 타입마다 다른 버전의 코드?
    → 소프트웨어 컴포넌트의 재사용성 개선
  - 클래스템플릿, 함수템플릿
- 함수템플릿 사용하기
  - 목적

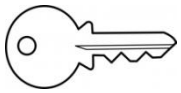
```
void main() {
  cout << "maximum(5, 3) = " << maximum(5, 3) << endl; //5
  cout << "maximum('x', 'y') = " << maximum('x', 'y') << endl;  //y
  cout << "maximum(3.3, 5.5) = " << maximum(3.3, 5.5) << endl ;  //5.5
}
```

C++의 다형성(polymorphism)을 이용

```
int maximum(int a, int b) {
    return a > b ? a : b;
}
char maximum(char a, char b) {
    return a > b ? a : b;
}
double maximum(double a, double b) {
    return a > b ? a : b;
}
```


함수템플릿을 이용

```
template <class T>
T maximum(T a, T b) {
    return a > b ? a : b;
}
```



스택 클래스

- 클래스 구현: ClassName.h, ClassName.cpp

- 선언과 정의의 차이
  - 함수의 선언(declaration): 프로토타입만 명시
  - 함수의 정의(definition): 구현(implementation). 함수의 선언에 함수의 몸체를 더한 형태

- 일반적인 경우(템플릿을 사용하지 않는 경우)
  - 함수의 선언은 헤더파일에
  - 함수의 정의는 소스파일에

- 스택 클래스: Stack.h, Stack.cpp, main.cpp
  - **실습: 코드를 작성해 보세요!!**

스택 클래스' - Stack.h

02.NontemplateStack

- Stack.h

```
class Stack {
private:
    int size;
    int top;
    int* stackPtr;
public:
    Stack(int size = 10);
    ~Stack() { delete[] stackPtr; }

    bool push(const int& item);
    bool pop(int& item);

    bool isEmpty() const { return (top == -1); }
    bool isFull() const { return (top == size-1); }
};
```

* Pass-by-references is more efficient than pass-by-value, because it does not copy the arguments. The formal parameter is an alias for the argument. When the called function read or write the formal parameter, it is actually read or write the argument itself.
* The difference between pass-by-reference and pass-by-value is that modifications made to arguments passed in by reference in the called function have effect in the calling function, whereas modifications made to arguments passed in by value in the called function can not affect the calling function. Use pass-by-reference if you want to modify the argument value in the calling function. Otherwise, use pass-by-value to pass arguments.
* The difference between pass-by-reference and pass-by-pointer is that pointers can be NULL or reassigned whereas references cannot. Use pass-by-pointer if NULL is a valid parameter value or if you want to reassign the pointer. Otherwise, use constant or non-constant references to pass arguments.

스택 클래스” - Stack.cpp

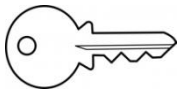
- Stack.cpp

```
#include "stdafx.h"
#include "Stack.h"

Stack::Stack(int s)
{
        size = s > 0 && s < 1000 ? s : 10 ;
        stackPtr = new int[size] ;
        top = -1 ;
}

bool Stack::push(const int& item)
{
        if (isFull()) return false;
        stackPtr[++top] = item ;
        return true;
}

bool Stack::pop(int& popValue)
{
        if (isEmpty()) return false;
        popValue = stackPtr[top--] ;
        return true;
}
```

스택 클래스‴ - main.cpp

- main.cpp

```
#include <iostream>
#include "Stack.h"
using namespace std;

void main()
{
    Stack stackInt;
    int i = 1;
    cout << endl << "stackInt에 원소들을 넣습니다:" << endl;
    while (stackInt.push(i)) { cout << i << ' '; i += 1; }
    cout << endl << "stackInt에서 원소들을 뺍니다:" << endl;
    while (stackInt.pop(i)) { cout << i << ' '; }
}
```



클래스템플릿

- 클래스템플릿의 정의
  - 클래스 내부 데이터 타입을 미리 가정하지 않고 그 클래스를 정의할 수 있음
- 선언과 정의 위치
  - 템플릿을 사용하지 않는 경우
    - 함수의 선언은 헤더파일에
    - 함수의 정의는 소스파일에
  - 템플릿의 경우
    - 클래스 템플릿의 멤버함수의 선언과 정의는 모두 같은 한 헤더파일에 있어야 함
    - 이유: 템플릿 멤버함수의 정의는 독립적으로 컴파일되지 못함
      - 실제로 사용될 타입의 객체를 생성하기 위해서는 그 시점에서 해당 템플릿의 정의 및 선언을 모두 찾을 수 있어야 함

- 템플릿 스택 클래스: Stack.h, main.cpp
  - **실습: 코드를 작성해 보세요!!**

클래스템플릿' – Stack.h

02.TemplateStack

- 클래스템플릿 선언 (Stack.h에 추가)

```
template <typename T>
class Stack {
private:
    int size;
    int top;
    T* stackPtr;
public:
    Stack(int size = 10);
    ~Stack() { delete[] stackPtr; }
    bool push(const T& item);
    bool pop(T& item);
    bool isEmpty() const { return (top == -1); }
    bool isFull() const { return (top == size-1); }
};
```

클래스템플릿" – Stack.h

- 클래스템플릿 정의 (Stack.h에 추가)

```
template <typename T>
Stack<T>::Stack(int s)
{
        size = s > 0 && s < 1000 ? s : 10 ;
        stackPtr = new T[size] ;
        top = -1 ;
}

template <typename T>
bool Stack<T>::push(const T& item)
{
        if (isFull()) return false;
        stackPtr[++top] = item ;
        return true;
}

template <typename T>
bool Stack<T>::pop(T& popValue)
{
        if (isEmpty()) return false;
        popValue = stackPtr[top--] ;
        return true;
}
```

클래스템플릿''' – main.cpp

- 클래스템플릿 사용하기 (main.cpp)

```
#include <iostream>
#include "Stack.h"
using namespace std;

void main() {
    Stack<float> stackFloat(5);
    float f = 1.1F;
    cout << "stackFloat에 원소들을 넣습니다:" << endl;
    while (stackFloat.push(f)) { cout << f << ' '; f += 1.1f; }
    cout << endl << "stackFloat에서 원소들을 뺍니다:" << endl;
    while (stackFloat.pop(f)) { cout << f << ' '; }

    Stack<int> stackInt;
    int i = 1;
    cout << endl << "stackInt에 원소들을 넣습니다:" << endl;
    while (stackInt.push(i)) { cout << i << ' '; i += 1; }
    cout << endl << "stackInt에서 원소들을 뺍니다:" << endl;
    while (stackInt.pop(i)) { cout << i << ' '; }
}
```

클래스 템플릿 인자

- 타입 인자뿐만 아니라 비타입 인자도 가질 수 있음

```
template <typename T, int size> class Stack { /* 클래스 정의 생략 */ };
```

  - 모든 클래스 템플릿 함수의 정의에 있어서 'template <class T>' 대신 'template <class T, int size>'를 사용해야 함
  - 'Stack<T>' 대신 'Stack<T, size>'를 사용
  - 클래스 내의 변수 'size'는 제거

- 템플릿 인자는 디폴트값을 가지도록 명시할 수 있음

```
template <typename T = float, int size = 100> class Stack { /* 클래스 정의 생략 */ };
```

  - 타입을 Stack<float, 100>이 아니라 Stack<>으로 선언하면 Stack<float, 100>으로 하였을 때와 동일하게 수행됨
  - 한 인자의 디폴트값을 명시하면 이후의 모든 인자들에 대해서도 디폴트값들을 명시해야 함

typedef

- typedef
  - 특히 템플릿의 사용에 있어서 매우 유용함
    - 난해해 보이는 템플릿 문법으로부터 해방
      - 마치 사용자정의 타입인듯
    - 템플릿의 구현부를 별도로 분리
      - 템플릿 구현부가 바뀌는 경우에는 typedef 문만 수정해주면 됨
  - 정수값들을 위한 템플릿벡터 'Stack<int>' 대신 'INTSTACK'

```
typedef Stack<int> INTSTACK ;
typedef Stack<float> FLOATSTACK;
```

```
Stack<int> istack;
Stack<float> fstack(5);
```

```
INTSTACK istack;
STACKFLOAT fstack(5);
```

템플릿의 instantiation

- 템플릿의 instantiation
  - 컴파일러는 instantiation을 의미하는 문을 만나는 때에 템플릿으로부터 클래스/함수를 생성함.
  - 예제

```
template <typename T>
class Stack
{
public:
  Stack();
 ~Stack();
 void f();
 void g();
};
```

```
int main()
{
 Stack<int> istack;
 istack.f() ;
 Stack<float> fstack;
 fstack.g() ;
 return 0 ;
}
```

    - 컴파일러는 Stack<int> 클래스, Stack<float> 클래스를 generate함.
      - 클래스를 generate할 때에는 항상 constucts/destructors들도 함께 생성함
    - 컴파일러는 Stack<int>::f() and Stack<float>::g() 의 definition도 generate함.
    - 컴파일러는 instantiation할 필요 없는 함수들의 definition들은 생성하지 않음.
      - 예제에서, Stack<int>::g()와 Stack<float>::f()의 definition들은 필요 없으므로 generate하지 않음.

상수화를 위한 const의 사용

- 경우 1: 일반 변수의 선언 시 사용
  - 해당 변수의 값을 수정할 수 없음을 의미
  - 한번 초기화된 후에는 변경이 불가능하므로 반드시 선언과 동시에 초기화시켜야 함

```
const int c = 5;
c = 2; //error
c++; //error
```

상수화를 위한 const의 사용'

- 경우 2: 포인터 변수의 선언 시 사용
  - 포인터 변수가 가리키는 곳의 변수값을 바꿀 수 없음을 의미함
    - 다른 변수의 주소값을 그 변수에 할당할 수는 있음
    - 따라서 변수의 선언 시에 꼭 초기화하지 않아도 됨

```
int c = 5;
const int* p;
*p = 5; //error
p = &c; //ok
```

  - 포인터 변수 자체를 상수화 시키려면 변수명 직전에 const 사용
    - 이 경우, 다른 주소값으로 수정할 수 없음
    - 따라서 반드시 초기화해야 함

```
int c = 5;
int* const p = &c;
*p = 5; //ok
p = &c; //error
```

상수화를 위한 const의 사용”

- 경우 2: 포인터 변수의 선언 시 사용
  - const를 두 번 사용도 가능
    - 포인터 변수 자체도 상수화시키고 가리키는 대상값도 상수화시킴
    - 반드시 초기화가 필요함

```
int c = 5;
const int* const p = &c;
*p = 5; //error
p = &c; //error
```

상수화를 위한 const의 사용‴

- 경우 3: 함수의 인자에 사용
  - 인자값을 함수 내에서 수정하지 않음을 의미
    - 실수로 인자값 변경을 방지함

```
void myfunc(const int c) { c = 2; } //error
```

- 경우 4: 클래스 멤버 함수의 선언 시 사용
  - 해당 함수 내부에서 어떤 멤버 변수의 값도 변경하지 않음을 의함

```
class MyClass  {
private:  int c;
public:   void myfunc() const { c = 2; } //error
};
```

- 경우 5: 함수의 선언 시 리턴 타입명에 사용
  - 리턴값을 받는 쪽에서 리턴값을 수정하지 못하도록 함

```
const int* myfunc() { /*생략*/ }
int* b = myfunc(); //error
```

빠른 인자값의 전달을 위한 const의 사용

- 인자값의 전달: 값에 의한 전달 pass-by-value
  - 인자 변수를 위한 메모리 확보 -> 거기에 넘길 인자값을 복사
    - 함수의 내부에서 인자 변수의 값을 수정할 수는 있음
    - 함수가 리턴된 후에는 인자 변수의 메모리 공간이 없어짐 (수정 내용도 사라짐)

```
void myfunc(int a) { a = 5; }

int c = 3;
myfunc(c);
printf("c = %d₩n", c); //c=3
```

빠른 인자값의 전달을 위한 const의 사용'

- 인자값의 전달: 참조에 의한 전달: pass-by-reference
  - 인자 변수가 인자값의 별명으로 처리됨
    - 인자 변수에 가해지는 변경은 실제 인자값의 메모리 영역에 그대로 적용됨

```
void myfunc(int &a) { a = 5; }

int c = 3;
myfunc(c);
printf("c = %d₩n", c); //c=5
```

  - 인자값의 경우뿐만 아니라 일반 변수의 경우에도 별명을 만드는 용도로 참조 변수의 사용이 가능함
    - 참조 변수는 원래 변수와 사실상 동일한 실체임

```
int c = 3;
int&d = c;
d = 5;
printf("c = %d₩n", c); //c=5
```

빠른 인자값의 전달을 위한 const의 사용”

- 인자값의 전달: 참조에 의한 전달: pass-by-reference
  - 인자값의 복사로 인한 비용을 줄이는 용도로 사용됨
    - 기본 타입(int 등)등 이외의 구조체 등의 경우에는 이 방식을 사용하자.

```
void myfunc(const int &a) { printf(”a = %d₩n”, a); }
```

  - 인자 변수가 참조 변수인 경우에 const를 명시하면 표현식의 경우도 허용됨

```
void myfunc(const int &a) { printf(”a = %d₩n”, a); } //ok
void myfunc(int &a) { printf(”a = %d₩n”, a); } //error

myfunc(c+1);
```

형변환

- 형변환: C/C++의 단점이면서 동시에 장점임
  - 가급적이면 사용하지 말자.
  - 어쩔 수 없다면 C++ 형변환 방식을 우선하여 사용하자.

- C언어의 고전적 형변환
  - "(타입명) 표현식": 예: (int) a
  - 문맥을 전혀 인식하지 않는 강제적인 변환임

- C++ 스타일의 형변환
  - "xxx_cast<타입명> (표현식)"

  - static_cast
  - const_cast
  - reinterpret_cast
  - dynamic_cast

형변환: static_cast

- static_cast
  - C 언어의 형변환과 동일한 의미의 형변환 능력을 가짐
    - 형변환 시에 타입 체크를 하지 않음
    - 문법적인 엄격함을 준수하여 코딩하기 위한 목적이 큼.

```
int i = 5;
double d = (double)i / 10;
double d = static_cast<double>(i) / 10;
```

형변환: static_cast

- 예시를 위한 가정

```
class MyClass { /*…*/ };
class MyChildClass : public MyClass { /*…*/ };
Void func(MyChildClass* mcc) { /*…*/ };
```

- static_cast
  - 암시적 변환을 되돌리기 위한 용도
    - 상속 계층 구조 상에서 위쪽 방향으로는 암시적으로 형변환이 자동으로 수행됨
    - 상속 계층 구조 상에서 아래 방향으로 되돌리기 위해서 static_cast를 사용함
      - 강제로 형변환하며 그 결과는 개발자 몫임
      - 주의: 부모 클래스가 가상(virtual)클래스인 경우에는 static_cast 대신 dynamic_cast를 사용해야 함.

```
MyChildClass* mcc = /*…*/;
MyClass* mc = mcc; //암시적 타입 변환
….
MyChildClass* mcc = static_cast<MyChildClass*>(mc);
```

형변환: const_cast

- const_cast
  - 표현식의 상수성(const)이나 휘발성(volatile)을 없애는 용도로 사용
    - 예: 'const int' 형의 값을 'int' 형의 값으로 바꾸고자 하는 경우

```
MyChildClass mcc;
const MyChildClass &cmcc = mcc;

func(&cmcc); //error: func은 비상수성 인자를 받는 함수임

func( const_cast<MyChildClass*>(&cmcc) ); //ok
//func((MyChildClas*) &cmcc); 의 C 스타일도 가능하지만 피하자
```

형변환: dynamic_cast

- dynamic_cast
  - 상속 계층 관계를 가로지르거나 하향된 클래스 타입으로 안전하게 형변환함
    - 즉 기본 클래스 객체의 포인터나 참조자 타입을 파생(derived) 클래스 또는 형제(sibling) 클래스의 타입으로 변환할 때에 사용됨

```
MyClass* mc = /*...*/ ;
func( dynamic_cast<MyChildClass*>(mc) ); //ok
```

    - 만약 mc가 MyChildClass 객체의 포인터가 아니라면 형변환 결과로 null 포인터가 리턴됨

  - 사용 제약
    - 상속 계층 구조를 따라 갈 때에만 사용해야 함
    - 가상 함수가 없는 타입에는 적용할 수 없음

형변환: reinterpret_cast

- reinterpret_cast
  - 임의의 포인터 타입끼리 변환하도록 지원함
    - 상속관계에 있지 않은 포인터끼리도 형변환이 가능함.

```
int* pi = reinterpret_cast<int*>(1234);

char* pc = reinterpret_cast<char*>(pi);
```

  - 모든 형변환을 허용하지만 그 책임은 모두 개발자 몫임
    - 강제 변환이므로 안전하지 않음
    - 최대한 피하자