C++ A-Z

Overloads and templates

선택한 이유: 자바 오버로딩 / 오버라이딩 개념만 기억나고, 이것마저도 가물가물해서 기억 환기할 필요성을 느낌.

함수 오버로딩

함수 이름이 같아도 매개변수 타입이나 개수가 다르면 별개의 함수로 인식되는 것.

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

int operate (int a, int b) { return (a*b); }
double operate(double a, double b) { return a / b; }

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

느낀 점: 정말 진짜 별 다를 것 없는 오버로딩 개념. 굳이 게임에 적용하자면 대미지 계산할 때 이용한다? 기본 대미지는 정수형으로 받고, 버프 계산이 적용되거나 스킬로 강화된 공격은 실수형으로 받아서 대미지 연산을 다르게 적용하는 메커니즘.

함수 템플릿

바디 내용이 동일한 오버로딩 케이스는 template을 이용해 통일이 가능함.
문서 예제의 sum(int, int)와 sum(double, double)은 template <classT> T sum(T a, T b)로 쓸 수 있다. 신기하네....

```
template <template-parameters> function-declaration
name <template-arguments> (function-arguments)
```

느낀 점: 타입 유연성, 중복 코드 제거 등의 장점이 있음. 위와 동일하게 대미지 계산이나 HP/MP 회복 등이 동일한 로직을 사용하고 있다면 템플릿으로 simplify 가능할 것 같음.

Auto-deduction / 타입 추론

편의 기능인 듯. 냅다 함수에 값을 넣으면 컴파일러가 type을 추론해줌. 와!
이때, 타입이 섞이면 안 된다.

그럼 처음부터 함수가 (str string, int integer) 같은 거면 어떡함.
ㄴ아... 다중 템플릿 타입을 써야 한다고 함.

```
template <class T>
T sum(T a, T b) { return a + b };  << 여기에 sum(10, 2.5) 이런 거 넣을 시 컴파일러가 에러 뱉음.
```

```
template <class T, class U>
auto sum(T a, U b) -> decltype(a + b) { return a + b; }  << auto + decltype(~)으로 컴파일러 자동사냥. 리턴타입 네가 알아서 해 줘.
```

다중 템플릿 파라미터. 다른 타입이어도 비교 가능하게 해줌.
`template <class T, class U> bool are_equal(T a, U b)`

그래서 이걸 어디에 쓰지....
ㄴ검색해 보니 입력 유효성 검사로 사용한다고 함.

Non-type 템플릿 인자

정수 상수를 고정해서 템플릿에 사용할 수 있다! 컴파일 타임에 결정되어 쭉 변함없이 사용되는 값이기 때문에 효율 좋음. 근데 왜 constant template arguments라고 안 부르는 거임. (추가: 검색해 봤더니 자료형(type)이 아니라 값(value)을 넘겨주는 argument이기 때문에 타입이 없다(X) 타입이 아닌 정해진 인자(O)라는 뜻으로 사용하는 듯;)

느낀 점: 이건 쓸 데 많을 듯. 장판 지속 시간에 비례한 대미지 계산이나 버프 지속 시간에 비례한 효과 적용 등....

```
template <class T, int N>
T atkBuff(T val) { return val * N; }
```

사용할 때는 atkBuff<int, 2>(50); 이런 식으로...?

Name Visibility 이름이 눈에 보이냐고???

선택한 이유: 제목을 봤는데 무슨 내용일지 감이 하나도 안 와서. 이름 시계가 뭘까.

Scope와 유효 범위

읽으니까 바로 감이 왔다. 변수, 함수 등이 어디에 선언되냐, 즉 위치에 따라 보여지는? 게 달라짐. Global이냐 Local이냐 차이겠지.... 오버라이딩 생각이 나.

전역 변수 - 함수 밖에 있음
지역 변수 - 함수 안에 있고 여기서만 쓸 거임.
밖과 안 동시에 존재하는 변수들은 가려지는? 거라고 함.

느낀 점 : 메모리 관리를 위해서 지역 변수를 자주 사용하게 되겠지만 게임 응용 방식으로 딱 떠오르는 게 없음. 플레이어의 상태나 시스템 설정 같은 것들은 전역 변수로 다뤄야 하겠지만 지역 변수는 워낙 다양해서 규정하기 어렵다. 턴제 전투의 에너미 HP 같은 건 지역 변수로 쓸 듯.

Block Scope & Hiding

Only one entity can exist with a particular name in a particular scope. 라고 함.
하나의 스코프 안에서 중복된 이름을 사용하게 되는 경우 에러 반환.

내부 블록에서는 외부 변수와 같은 이름을 쓸 수 있는데, 이때 이건 숨겨진 상태라고 한다. Encapsulation 개념이었던 것 같은데 확인 필요. (추가: 스코프 기반 name hiding/이름 가리기라고 함. 언어 차원에서 제공하는 문법적 동작이라고 함.)

```
int x = 10;

int main() {
	{
		int x = 20;
		cout << x << endl; // x = 20
	}
	cout << x << endl; // x = 10
}
```

느낀 점: 이름 짓기는 쉬워져도 코드 나중에 보면 무슨 말인지 모를 듯. 주석을 열심히 달도록 합시다.

네임스페이스(namespace)

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

지역 변수야 짧으니까 대부분 문제 없이 작동하지만 전역변수의 경우 여러 함수와 얽혀 있고 코드 전반에 걸쳐 사용되므로 이름 충돌에 상대적으로 더 취약함. 이것을 방지하기 위해 네임스페이스를 이용함.

namespace identifier { named_entities }
myNamespace::a
myNamespace::b

예시)
namespace player { int hp = 100, int mp = 50 }
namespace enemy { int hp = 300; int mp = 80; }
hp랑 mp 변수 이름이 같지만 namespace가 달라서 충돌X

```
int main() {
	cout << "Player HP: " << Player::hp << endl;
	cout << "Enemy HP: " << Enemy::hp << endl;
}
```

cout과 <<가 함께 사용되는 경우 - insertion operator로 작동
콘솔 출력 스트림에 “~“ 문자열을 추가, 변수 값 추가, 줄바꿈, 출력 버퍼 flushing

array 느낌으로 묶어두고 꺼내서 쓰는 느낌인 듯? 찾아보니 이름을 가진 데이터를 그룹화한 것이지 array는 아니라고 함. namespace는 index도 없네….

Using (namespace)

using으로 불러온 이름(변수)는 temporarily 고정되어서 다른 namespace의 같은 이름을 가진 변수를 불러오기 전까지는 이름만 가지고 사용해도 불러온 이름을 사용하는 개념?

using player::hp; 는 하나만 고정시키는 거고,
using namespace player ;는 player 네임 스페이스 안의 모든 변수를 고정시킴.

```
namespace player { int hp = 100, int mp = 50 }
namespace enemy { int hp = 300; int mp = 80; }

int main() {
	using player::hp;
	using player::mp;
	using namespace player; // 위 두 줄과 동일.
	cout << "Player HP: " << hp << endl;
	cout << "Player MP: " << mp << endl;
}
```

느낀 점: 어떤 캐릭터/엔티티 선택했는지에 따라 automatically 출력이 달라지는 코드 짤 수 있을 듯. 편리하다.

Namespace Alias

네임스페이스 이름이 길다? 축약해서 사용 가능하게 만들어줌.
namespace php = player::hp; 같은 식으로….

코더의 편의성을 생각한 기능으로 보임. 근데 축약어 남발하면 헷갈릴 듯. 그리고 다른 코더들이랑 협업할 때 주석없이 alias 사용하면 몰매 맞을 것. 보통 메모장이나 엑셀 같은 곳에 축약어 저장을 해두나? 어떻게 팀프로젝트에서 합의를 보는 거지.

std 네임스페이스

표준 라이브러리의 모든 기능은 이 std 네임스페이스 안에 존재. 그래서 코드 앞에 using namespace std;를 써서 고정을 해주는 것. 이제 일일이 std::어쩌구 변수 안 붙여도 사용할 수 있다!!! 대박대박.

없으면 cout도 std::cout처럼 써줘야 하는 불상사 발 생.

Storage Class (저장 클래스??)

읽어보니 전역 변수는 static storage에 넣어서 프로그램이 실행되는 동안 계속 유지될 수 있게 하고, 지역 변수는 automatic storage에 넣어서 함수가 끝나면 사라지게 만드는 메모리 관리 클래스 말하는 것이었다.

```
// static vs automatic storage
#include <iostream>
using namespace std;

int x;

int main ()
{
          int y;
          cout << x << '\n';
          cout << y << '\n';
          return 0;
}
```

int x는 0으로 초기화, y는 쓰레기 값이 됨.

Friendship의 개념

우정이 뭔데. 친구가 뭔데. 바로바로 private/protected 클래스 멤버를 외부에서 접속할 수 있게 만들어 주는 선언. 친구(friend)라면 외부 함수 및 클래스가 친한 척할 수 있다.

```
class Rectangle {
        int width, height;
        friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param) {
        // 여기서 private 멤버 접근 가능!
}
```

근데 이게 단방향 우정이지 쌍방은 아님. 슬픈데.
+) 비전이성 : 친구의 친구는 친구가 아니다. 웃긴데.
그리고 이건 상속과는 다른 개념.