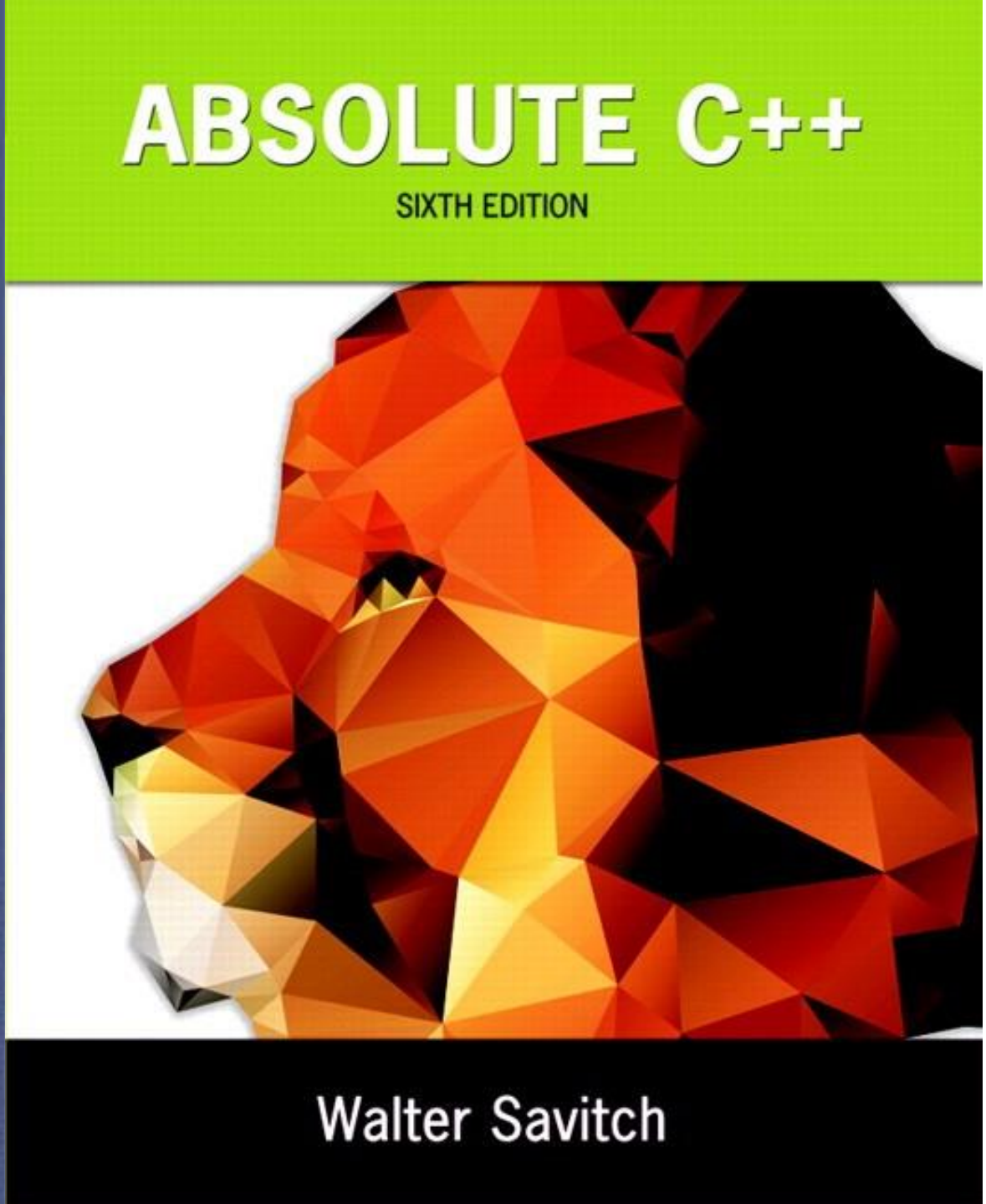

Chapter 16

Templates

PEARSON

Introduction

- C++ templates
  - 함수들과 클래스들에 대해서 매우 일반적인 정의들을 허용함
  - 데이터 타입 이름들이 파라미터들로 전달됨
  - 구체적인 데이터 타입과 이것에 의존적인 클래스와 함수들의 정의는 런타임에 결정됨

Function Templates

- 다음 함수를 살펴보자:

```
void swapValues(int& var1, int& var2)
{
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- 이 함수는 오직 int 타입의 변수들만 허용한다.
- 그러나, 코드 자체는 아무 타입에나 적용할 수 있다.

Function Templates vs. Overloading

- char 타입을 사용하길 원하면 오버로딩 함수를 다음과 같이 정의할 수 있다:

```
void swapValues(char& var1, char& var2)
{
    char temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- 그러나, 코드는 변경 사항이 없다.

Function Template Syntax

- 아무 타입의 변수들을 받아주는 swapValues 함수:

```
template<class T>
void swapValues(T& var1, T& var2)
{
        T temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- 첫번째 라인은 "template prefix " 라고 부른다.
  - 컴파일러에게 템플릿이 시작됨을 알린다.
  - 컴파일러에게 T 가 타입 파라미터임을 알린다.

Template Prefix

- 다음 template prefix 를 다시 살펴보자:
  template<class T>
- 이것은 class 타입만 받아들인다는 의미는 아니고, 아무 타입이나 받아들인다는 의미이다.
  - C++ 은 키워드 " class " 대신에 키워드 " typename " 을 사용할 수 있다.
  - 그러나, 대부분의 사람들은 "class" 키워드를 사용한다.
  - T 라는 파라미터 이름을 다른 단어로 바꿀 수 있다.

Calling a Function Template

- 다음 함수를 호출한다고 가정해보자:
  swapValues(int1, int2);
  - C++ 컴파일러는 템플릿을 이용하여 두 개의 int 파라미터를 위한 함수 정의를 컴파일 단계에서 이미 생성하였다.
- 함수 호출 관점에서 템플릿 함수 호출시 특별히 해주어야 할 다른 조치는 없다.

Another Function Template

- 템플릿 함수 선언:

```
Template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

- 템플릿 함수 정의:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
      cout << stuff1 << endl
                 << stuff2 << endl
                 << stuff3 << endl;
}
```

showStuff Call

- 다음 함수 호출을 생각해보자:
  showStuff(2, 3.3, 4.4);
- 컴파일러는 템플릿 함수 정의를 이미 생성할 수 있다.
  - 두번째 인자가 double 이므로, T 를 double 타입으로 변경해준다.
- 함수 호출 결과는 다음과 같다:
  2
  3.3
  4.4

Compiler Complications

- 함수 선언과 정의
  - 선언과 정의를 대부분의 경우 분리한다. 선언은 헤더 파일에, 정의는 cpp 파일에.
  - 템플릿의 선언과 정의를 분리하는 것은 대부분의 컴파일러들이 지원하지 않는다.
- 템플릿 함수는 그 함수가 호출되는 소스 파일 안에 위치시키는 것이 가장 안전하다.
  - 종종 #include 디렉티브를 통해서 템플릿 정의들을 불러오기도 한다.

More Compiler Complications

- 당신의 컴파일러가 특별한 요구사항을 갖고 있는지 확인하라.
  - 어떤 컴파일러는 특수 옵션들을 설정할 것을 요구한다.
  - 어떤 컴파일러는 템플릿 정의들과 다른 아이템들의 정의 간에 특별한 순서를 요구한다.
- 사용성이 가장 높은 프로그램 배치:
  - 템플릿 정의는 그것을 호출하는 소스 파일 안에 위치시킴
  - 템플릿 정의는 그것을 호출하는 모든 코드들에 선행하도록 위치시킴 (#include 디렉티브를 사용할 수 있음)

Multiple Type Parameters

- 다음 템플릿 선언도 가능하다:
  template<class T1, class T2>
- 그러나 일반적이지는 않다
  - 대개의 경우 하나의 타입만 교체하기를 원한다.
  - 사용되지 않는 타입 파라미터는 열거하면 안 된다.

Algorithm Abstraction

- 알고리즘들을 "일반적인" 방식으로 표현해라.
  - 알고리즘은 아무 타입의 변수들에게 적용된다.
  - 알고리즘의 본질적인 부분들에 집중하라.
- 함수 템플릿은 C++ 언어가 알고리즘 추상화를 지원하는 한 가지 방식이다.

Defining Templates Strategies

- 특정 타입을 전제로 함수를 개발하라.
- 그 함수를 완전히 bug-free 하게 만들어라.
- 그리고나서 템플릿 함수로 변경해라.
- 장점:
  - 구체적인 경우를 해결하는 것이 더 쉽다.
  - 알고리즘을 다루는 것이 먼저이고, 템플릿 문법은 나중에 다룬다.

Inappropriate Types in Templates

- 템플릿 코드가 의미가 통하는 한, 아무 타입이나 템플릿에 파라미터로 전달할 수 있다.
- 예를 들어, swapValues() 템플릿은,
  - 할당 연산자(=)가 정의되지 않는 타입들에게는 사용될 수 없다.
  - 예: int a[10], b[10];
    swapValues(a, b);

Class Templates

- 클래스도 템플릿으로 일반화할 수 있다.
  template<class T>
  - 클래스 정의 앞에 붙인다.
  - 클래스 정의 안에 모든 "T" 인스턴스들은 타입 파라미터로 교체된다.
- 일단 템플릿이 정의되면, 그 클래스의 객체를 선언할 수 있다.

Class Template Definition

- ```
  template<class T>
  class Pair
  {
  public:
        Pair();
        Pair(T firstVal, T secondVal);
        void setFirst(T newVal);
        void setSecond(T newVal);
        T getFirst() const;
        T getSecond() const;
  private:
        T first; T second;
  };
  ```
  ```

Template Class Pair Members

- ```
  template<class T>
  Pair<T>::Pair(T firstVal, T secondVal)
  {
      first = firstVal;
      second = secondVal;
  }
  template<class T>
  void Pair<T>::setFirst(T newVal)
  {
      first = newVal;
  }
  ```
  ```

Template Class Pair

- 이 템플릿 클래스의 객체는 타입 T 의 값들의 쌍을 갖는다.
- 다음과 같이 객체를 선언할 수 있다:

```
Pair<int> score;
Pair<char> seats;
```

- 다음과 같이 멤버 함수를 호출할 수 있다:

```
score.setFirst(3);
score.setSecond(0);
```

Pair Member Function Definitions

- 멤버 함수 정의에서 유의할 점들:
  - 각 정의는 그 자체가 템플릿 함수이다
  - 각 정의 앞에 template prefix 가 요구된다
  - :: 앞에 클래스 이름은 "Pair<T> " 라고 써야 한다
  - 그러나, 생성자 이름과 소멸자 이름은 그냥 "Pair" 와 "~Pair" 라고 써야 한다.

Class Templates as Parameters

- 템플릿 타입들은 표준 타입들이 쓰이는 어느 곳에서나 사용될 수 있다:

```
int addUP(const Pair<int>& the Pair);
```

Class Templates Within Function Templates

- 오버로딩 함수 대신에 템플릿 함수를 사용할 수 있다:

```
template<class T>
T addUp(const Pair<T>& thePair);
//Precondition: Operator + is defined for values
                of type T
//Returns sum of two values in thePair
```

- 위 addUp 함수는 모든 종류의 T 에 대해서 사용할 수 있게 된다.

Restrictions on Type Parameter

- 오직 "reasonable" 타입들만 T 를 대신할 수 있다.
  - 할당 연산자(=)가 잘 동작해야 한다.
  - 복사 생성자 또한 잘 동작해야 한다.
  - 타입 T 가 멤버 포인터들을 가질 때에는 소멸자가 적절하게 작성되어야 한다.

Type Definitions

- 특정 T 타입을 적용한 템플릿 클래스를 위해서 타입 이름을 정의할 수 있다.
- 예:

```
typedef Pair<int> PairOfInt;
PairOfInt pair1, pair2;
```

- PairOfInt 라는 타입 이름은 또한 파라미터로 사용될 수 있다.

Friends and Templates

- 템플릿 클래스들도 프렌드 함수들을 가질 수 있다.
  - 일반적인 클래스가 프렌드 함수를 갖는 것과 동일하다.
  - 연산자 오버로딩에서 특히 많이 사용된다.

Predefined Template Classes

- 벡터 클래스는 템플릿 클래스이다.
  - vector<int> A;
- basic_string 클래스는 템플릿 클래스이다.

basic_string<char>	works for char's
basic_string<double>	works for doubles
basic_string<YourClass>	works for YourClass objects

basic_string Template Class

- basic_string 클래스를 우리는 이미 사용하고 있다 → string 클래스는 basic_string<char> 로 정의한다.
- basic_string 클래스는 library <string> 에 들어 있다.
  - std 네임스페이스 안에서 정의된다.

Templates and Inheritance

- 상속 관점에서 템플릿이 특별히 다르게 취급될 것은 없다.
  - 상속 문법은 동일하게 적용된다.
- 파생된 템플릿 클래스들
  - 템플릿 클래스 또는 비템플릿 클래스로부터 파생될 수 있다.
  - 파생된 클래스는 당연히 템플릿 클래스이다.