C++ Template

임종우 (Jongwoo Lim)

Generic Programming

Generic programming is a style of computer programming in which algorithms are written in terms of **to-be-specified-later** types that are then instantiated when needed for specific types provided as parameters.[wikipedia]

- C++ Standard Template Library (STL).
- Data containers such as matrix, vector, array, image, etc.
- Algorithms such as sorting, searching, hashing, etc.
- ...

Generic Programming

```
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Generic Programming

```
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// We also want to sort a double array.

void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Generic Programming

```
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// We also want to sort a double array.

void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}

// And also a string array.

void SelectionSort(string* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    string tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

C++ Template

- C++ template allows us to avoid this repeated codes.

```
// Suppose we want to sort an array of type T.

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

C++ Template

```
// Suppose we want to sort an integer array.

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
int main() {
  int array[] = { 2, 5, 3, 1, 4 };
  const int size = sizeof(array) / sizeof(int);
  SelectionSort<int>(array, size);    // You may use SelectionSort(array, size);
  for (int i = 0; i < size; ++i) cout << " " << array[i];
  cout << endl;
  return 0;
}
```

C++ Template

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

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    Swap(array[i], array[min_idx]);   // Clearly states the meaning of operation.
  }
}
```

C++ Template

- Functions and classes can be templated.
- Template parameters can be typenames (= classes) or integers.

```
template <class First, class Second>  // Same as <typename First, typename Second>.
struct Pair {
  First first;
  Second second;
};

template <typename T, int d>   // d must be a constant integer.
void Reverse(T array[d]) {     // Same as (T* array) - array size is not checked.
  for (int i = 0; i < d / 2; ++i) Swap(array[i], array[ d - i - 1]);
}

int main() {
  int array[10] = { ... };
  int size = 10;
  Reverse<int, 10>(array);     // OK.
  Reverse<int, size>(array);   // Error.
  return 0;
}
```

C++ Template

```
template <class First, class Second>
struct Pair {
  First first;
  Second second;

  Pair(const First& f, const Second& s) : first(f), second(s) {}
};

template <class First, class Second>
Pair<First, Second> MakePair(const First& first, const Second& second) {
  return Pair<First, Second>(first, second);
}

int main() {
  Pair<int, int> p = MakePair(10, 10);   // Equivalently MakePair<int, int>(10, 10);
  Pair<int, int> q = Pair<int, int>(20, 20);
  return 0;
}
```

C++ Template Classes

```
template <typename T, int d>
class Vector {
 public:
  typedef T DataType;     // Access as Vector<T, d>::DataType.

  Vector() {  for (int i = 0; i < d; ++i) vec_[i] = T(); }
  Vector(const Vector& v) {  for (int i = 0; i < d; ++i) vec_[i] = v.vec_[i]; }
  const int size() const { return d; }

  const T& operator[](int i) const { return vec_[i]; }
  T& operator[](int i) { return vec_[i]; }

  Vector operator+() const { return *this; }
  Vector operator-() const;

  T Sum() const;
  T Dot(const Vector& v) const;

 private:
  T vec_[d];
};
```

C++ Template Classes

```
template <typename T, int d>
Vector<T, d> Vector<T, d>:: operator-() const {
  Vector<T, d> ret;
  for (int i = 0; i < d; ++i) ret.vec_[i] = -vec_[i];
  return ret;
}

template <typename T, int d>
T Vector<T, d>::Sum()  const {
  T ret = T();
  for (int i = 0; i < d; ++i) ret += vec_[i];
  return ret;
}

template <typename T, int d>
T Vector<T, d>::Dot(const Vector& v)  const {
  T ret = T();
  for (int i = 0; i < d; ++i) ret += vec_[i] * v.vec_[i];
  return ret;
}
```

C++ Template Classes

```
template <typename T, int d>
class Vector {
 public:
  // ....

  template <typename S>
  Vector<S, d> cast()  const {
    Vector<S, d> ret;
    for (int i = 0; i < d; ++i) ret[i] =  static_cast<S>(vec_[i]);
    return ret;
  }

 private:
  T vec_[d];
};
```

```
int main() {
  Vector<int, 3> v, w;
  Vector<int, 3>::DataType dot = v.Dot(-w);
  Vector<double, 3> x = v.cast<double>();
  cout << x.Sum();
  return 0;
}
```

C++ Template Classes

```
int main() {
  Vector<int, 3> v;
  return 0;
}
```

```
template <>
class Vector<int, 3> {
 public:
  typedef int DataType;     // Access as Vector<T, d>::DataType.

  Vector() { for (int i = 0; i < 3; ++i) vec_[i] = int(); }
  Vector(const Vector& v) { for (int i = 0; i < 3; ++i) vec_[i] = v.vec_[i]; }
  const int size() const { return d; }

  const int& operator[](int i) const { return vec_[i]; }
  int& operator[](int i) { return vec_[i]; }

  Vector operator+() const { return *this; }
  Vector operator-() const;

  int Sum() const;
  int Dot(const Vector& v) const;

 private:
  int vec_[3];
};
```

Inline Function

- Request the compiler to insert the function body in the place that the function is called.
  - The function body will not be included in the object file.
- Compilers are not obligated to respect this request.
- Member functions defined in the class definition are inlined.
- Inline function definitions are placed in header files.
- Pros/cons: eliminate function call overhead / code bloat.

```
inline void Swap(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```