Template classes have many uses, but the most obvious one is to provide adaptable storage containers.

Suppose we want to create objects that can store two values, an integer and a character. For this purpose we could define

class pair\_int\_char {

public:

int first;

char second;

pair\_int\_char(int x, char y) : first(x), second (y) { }

};

pair\_int\_char pair1(13, ‘a’);

cout << pair1.first << endl;

cout << pair1.second << endl;

If we also want objects that can store, say, a boolean value and a double-precision floating-point number, we could define

class pair\_bool\_double {

public:

bool first;

double second;

pair\_bool\_double(bool x, double y) : first(x), second(y) { }

};

pair\_bool\_double pair2(true, 0.1);

cout << pair2.first << endl;

cout << pair2. second << endl;

The same can be repeated for any of the other infinitely many pairs of types, but a template class permits them all to be expressed with a single definition:

template <class T1, class T2>

class pair {

public:

T1 first;

T2 second;

pair(T1 x, T2 y) : first(x), second(y) { }

};

WRITING

pair<int, char> pair3(13,’a’);

pair<bool, double> pair4(true, 0.1);

declares pair3 and pair4 with structure equivalent to pair1 and pair2, respectively.

Types defined by other classes can be used as the actual types; for instance,

pair<pair\_int\_char, float> pair7(pair1, 1.23);

or, equivalently,

pair(pair<int, char>, float> pair8(pair3, 1.23);

An improvement can be made in template class definition by passing x and y as constant reference parameters instead of passing them by value so that less memory is wasted.

template <class T1, class T2>

class pair {

public:

T1 first;

T2 second;

pair(const T1& x, const T2& y) : first(x), second(y) { }

};