# C++ Templates

CSCI 1061U — Programming Workshop 2

**faisal.qureshi@uoit.ca**
**Faculty of Science**
**University of Ontario Institute of Technology**

# Without templates

```
#include <iostream>

int min(int a, int b)
{
  if (a < b) return a;
  return b;
}

int main()
{
  using std::cout;
  using std::endl;

  cout << min(1,2) << endl;

  return 0;
}
```

# Using templates

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
  if (a < b) return a;
  return b;
}

int main()
{
  using std::cout;
  using std::endl;

  cout << min<int>(1,2) << endl;
  cout << min<float>(1.4,2.2) << endl;

  return 0;
}
```

# Without templates

```
#include <iostream>

void swap(int& i, int& j)
{
    int temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  int i = 1, j = 2;
  swap(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

# Using templates

```
#include <iostream>

template <typename T>
void swap(T& i, T& j)
{
    T temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  int i = 1, j = 2;
  swap<int>(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

Works for int

# Without templates

```
#include <iostream>

void swap(int& i, int& j)
{
    int temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  int i = 1, j = 2;
  swap(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

# Using templates

```
#include <iostream>

template <typename T>
void swap(T& i, T& j)
{
    T temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  float i = 1, j = 2;
  swap<float>(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

Now works for floats!

# Without templates

```
#include <iostream>

void swap(int& i, int& j)
{
    int temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  int i = 1, j = 2;
  swap(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

# Using templates

```
#include <iostream>

template <typename T>
void swap(T& i, T& j)
{
    T temp = i;
    i = j;
    j = temp;
}

int main()
{
  using std::cout;
  using std::endl;

  SuperHero i(“Wolverine”,3);
  SuperHero j(“Batman”,1.1);
  swap<SuperHero>(i, j);
  cout << "i=" << i << endl;
  cout << "j=" << j << endl;

  return 0;
}
```

Now works for SuperHeros!

```
#include <iostream>

int sum(int a[], int n)
{
  int sum = 0;
  for (int i=0; i<n; ++i) {
    sum += a[i];
  }
  return sum;
}

int main()
{
  using std::cout;
  using std::endl;

  int a[] = {1, 2, 3};
  int arr_sum = sum(a, 3);

  cout << "sum = " << arr_sum << endl;

  return 0;
}
```

# Templatize

So that it can compute sum of arrays of types other than int

## Without templates

```
int sum(int a[], int n)
{
  int sum = 0;
  for (int i=0; i<n; ++i) {
    sum += a[i];
  }
  return sum;
}
```

*Invocation*

```
int a[] = {1, 2, 3};
int arr_sum = sum(a, 3);
```

Only works for int arrays

## Using templates

```
template <typename T>
T sum(T a[], int n)
{
  T sum = 0;
  for (int i=0; i<n; ++i) {
    sum += a[i];
  }
  return sum;
}
```

*Invocation*

```
float a[] = {1., 2.6, 3};
int arr_sum = sum<float>(a, 3);
```

Works for arrays of any type as long as (+=) and assignment is valid

# Without templates

```
#include <iostream>

void prn(int i, float j, char k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn(1,2.5,'A');

  return 0;
}
```

# Using templates

```
#include <iostream>

template <typename K, typename L, typename M>
void prn(K i, L j, M k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn<int, float, char>(1,2.5,'A');
  prn<int, char, float>(1,'A',3.2);

  int n=1;
  prn<int, int*, char>(1,&n,'A');


  return 0;
}
```

# Without templates

```
#include <iostream>

void prn(int i, float j, char k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn(1,2.5,'A');

  return 0;
}
```

# Using templates (Type-safe)

```
#include <iostream>

template <typename K, typename L, typename M>
void prn(K i, L j, M k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn<int, float, char>(1,2.5,'A');
  prn<int, char, float>(1,'A',3.2);

  float n=1;
  prn<int, int*, char>(1,&n,'A');


  return 0;
}
```

Doesn't work. float* is not the same as int*

# Without templates

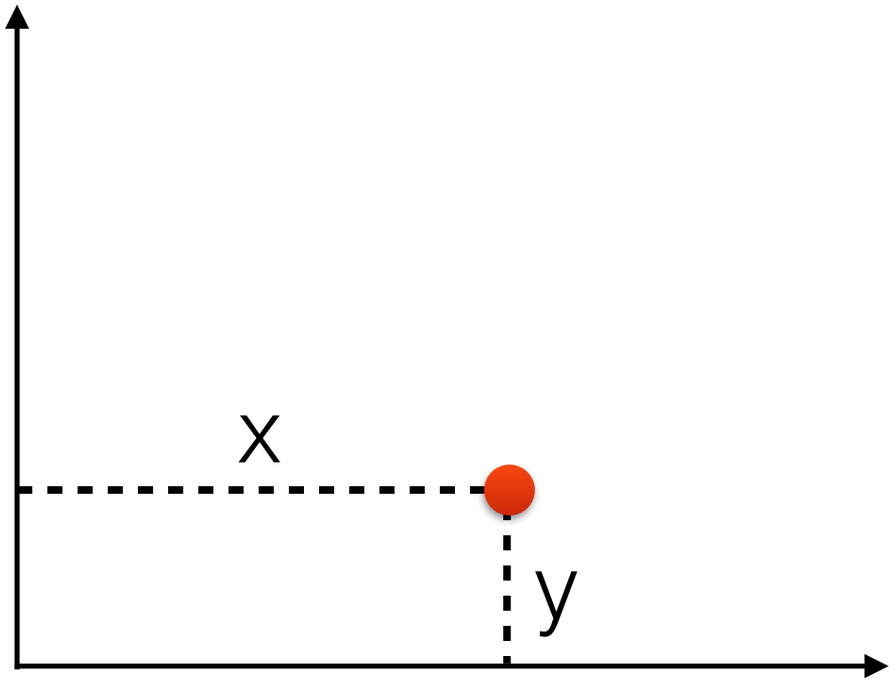
```
#include <iostream>

void prn(int i, float j, char k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn(1,2.5,'A');

  return 0;
}
```

# Using templates (Type-safe)

```
#include <iostream>

template <typename K, typename L, typename M>
void prn(K i, L j, M k)
{
  using std::cout;
  using std::endl;

  cout << i << endl;
  cout << j << endl;
  cout << k << endl;
}

int main()
{
  prn<int, float, char>(1,2.5,'A');
  prn<int, char, float>(1,'A',3.2);

  float n=1;
  prn<int, float*, char>(1,&n,'A');


  return 0;
}
```

Now works!

# A 2D point class




```
#include <iostream>
using std::ostream;
using std::cout;
using std::endl;

class pt
{
  protected:
  float _x, _y;

  public:
  pt(float x, float y)
    : _x(x), _y(y)
    {}

  float& x() { return _x; }
  float& y() { return _y; }
  const float& x() const { return _x; }
  const float& y() const { return _y; }
};

int main()
{
  pt a(1.0, 2.5);

  return 0;
}
```

# Without templates

```
class pt
{
  protected:
  float _x, _y;

  public:
  pt(float x, float y)
    : _x(x), _y(y)
    {}

  float& x() { return _x; }
  float& y() { return _y; }
  const float& x() const { return _x; }
  const float& y() const { return _y; }
};
```

# Using templates

```
template <typename T>
class pt
{
  protected:
  T _x, _y;

  public:
  pt(T x, T y)
    : _x(x), _y(y)
    {}

  T& x() { return _x; }
  T& y() { return _y; }
  const T& x() const { return _x; }
  const T& y() const { return _y; }
};
```

# Without templates

```
class pt
{
  protected:
  float _x, _y;

  public:
  pt(float x, float y)
    : _x(x), _y(y)
    {}

  float& x() { return _x; }
  float& y() { return _y; }
  const float& x() const { return _x; }
  const float& y() const { return _y; }

  float d2();
};

float pt::d2()
{
  return _x*_x + _y*_y;
}
```

# Using templates

```
template <typename T>
class pt
{
  protected:
  T _x, _y;

  public:
  pt(T x, T y)
    : _x(x), _y(y)
    {}

  T& x() { return _x; }
  T& y() { return _y; }
  const T& x() const { return _x; }
  const T& y() const { return _y; }

  T d2();
};

template <typename T>
T pt<T>::d2()
{
  return _x*_x + _y*_y;
}
```

# Without templates

```
class pt
{
  protected:
  float _x, _y;

  public:
  pt(float x, float y)
    : _x(x), _y(y)
    {}

  float& x() { return _x; }
  float& y() { return _y; }
  const float& x() const { return _x; }
  const float& y() const { return _y; }

  friend ostream& operator<<(ostream& os, const pt& p);
};

ostream& operator<<(ostream& os, const pt& p)
{
  os << "(" << p.x() << "," << p.y() << ")";

  return os;
}
```

# Using templates

```
template <typename T>
class pt
{
  protected:
  T _x, _y;

  public:
  pt(T x, T y)
    : _x(x), _y(y)
    {}

  T& x() { return _x; }
  T& y() { return _y; }
  const T& x() const { return _x; }
  const T& y() const { return _y; }

  template<typename U>
  friend ostream& operator<<(ostream& os, const pt<U>& p);
};

template <typename K>
ostream& operator<<(ostream& os, const pt<K>& p)
{
  os << "(" << p.x() << "," << p.y() << ")";

  return os;
}
```

Hmmm… not T

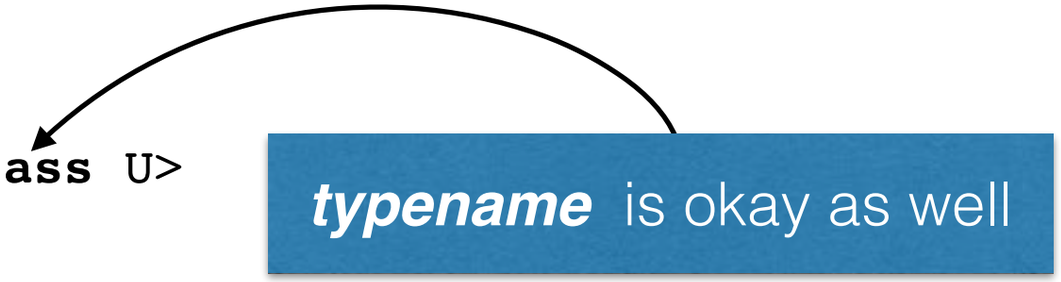
## Without templates

```
int main()
{
  pt a(1.0, 2.5);
  cout << a.d2() << endl;
  cout << a << endl;
  return 0;
}
```

## Using templates

```
int main()
{
  pt<float> a(1, 2.5);
  cout << a.d2() << endl;
  cout << a << endl;
  return 0;
}
```

# A class to store a pair of two different items




```
template<class T, class U>
class Pair
{
public:
    Pair() {}
    Pair(const T& firstValue, const U& secondValue);

    void setFirst(const T& newValue);
    void setSecond(const U& newValue);

    const T& getFirst( ) const;
    const U& getSecond( ) const;

private:
    T first;
    U second;
};
```

# A class to store a pair of two different items

```
template<class T, class U>
Pair<T,U>::Pair(const T& firstValue, const U& secondValue)
{
    first = firstValue;
    second = secondValue;
}

template<class T, class U>
void Pair<T,U>::setFirst(const T& newValue)
{
    first = newValue;
}

template<class T, class U>
const T& Pair<T, U>::getFirst() const
{
    return first;
}

template<class T, class U>
void Pair<T, U>::setSecond(const U& newValue)
{
    second = newValue;
}

template<class T, class U>
const U& Pair<T,U>::getSecond() const
{
    return second;
}
```

# A class to store a pair of two different items

```
int main( )
{
    Pair<char, int> p('A', 12);

    cout << "First is " << p.getFirst() << endl;
    p.setFirst('Z');
    cout << "First changed to " << p.getFirst() << endl;

    cout << "Second is " << p.getSecond() << endl;
    p.setSecond(2234);
    cout << "Second changed to " << p.getSecond( ) << endl;

    return 0;
}
```

# A class to store a pair of two different items

Note: we didn't have to declare this a friend of Pair class. Why?

```
template<class T, class U>
ostream& operator<<(ostream& os, const Pair<T,U>& p)
{
  os << "[" << endl
     << "1: " << p.getFirst() << endl
     << "2: " << p.getSecond() << endl
     << "]";
  return os;
}


int main( )
{
    Pair<char, int> p('A', 12);
    cout << p << endl;

    return 0;
}
```

We overload the << operator for our Pair class. Now we can do the following.

# A class to store a pair of two different items

This space is important and can induce crying in grown, otherwise well-adjusted, individuals

Nesting

```
int studentnumber = 10032120;
Pair<string, float> name_gpa("John",2.3);
Pair<int, Pair<string, float> > record(studentnumber, name_gpa);
```

**record**
Pair<int, Pair<string, float> >

**first**
int

**second**
Pair<string, float>

**first**
string

**second**
float

# A class to store a pair of two different items

Use ***typedef*** to define new types

```
int studentnumber = 10032120;
Pair<string, float> name_gpa("John",2.3);
Pair<int, Pair<string, float> > record(studentnumber, name_gpa);
```

||

```
typedef Pair<string, float> ng;
typedef Pair<int, ng> rng;

rng record1(studentnumber, ng("John", 2.3));
```

# Specialization

```
template <typename T>
class loc
{
  protected:
  T _x, _y;

  public:
  loc() {}
  loc(T x, T y)
    : _x(x), _y(y)
  {}

  T x() const { return _x; }
  T y() const { return _y; }

  void multiplyBy2();
};
```

```
template <typename T>
void loc<T>::multiplyBy2()
{
  _x *= 2;
  _y *= 2;
}
```

```
loc<float> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;

loc<double> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;

loc<int> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;
```

This function is called no matter what type T is

# Specialization

```
template <typename T>
class loc
{
  protected:
  T _x, _y;

  public:
  loc() {}
  loc(T x, T y)
    : _x(x), _y(y)
  {}

  T x() const { return _x; }
  T y() const { return _y; }

  void multiplyBy2();
};
```

```
loc<float> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;

loc<double> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;

loc<int> l(1.2, 3.4);
l.multiplyBy2();
cout << l.x() << ", " << l.y() << endl;
```

Specialization for type int

```
template <typename T>
void loc<T>::multiplyBy2()
{
  _x *= 2;
  _y *= 2;
}
```

```
template <>
void loc<int>::multiplyBy2()
{
  _x = (_x << 2);
  _y = (_y << 2);
}
```

# Dynamic Allocation

```
#include <iostream>
using std::cout;
using std::endl;

template<class T>
T* create_array(int n)
{
  T* a = new T [n];
  return a;
}

int main()
{
  double* a = create_array<double>(5);
  for (int i=0; i<5; ++i) {
    a[i] = i * 2;
  }

  for (int i=0; i<5; ++i) {
    cout << a[i] << endl;
  }

  delete[] a;

  return 0;
}
```

Allocate an array of double

# Dynamic Allocation

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

template<class T>
T* create_array(int n)
{
  T* a = new T [n];
  return a;
}

int main()
{
  string* movies = create_array<string>(2);
  movies[0] = "Casablanca";
  movies[1] = "On the Waterfront";

  for (int i=0; i<2; ++i) {
    cout << movies[i] << endl;
  }

  delete[] movies;

  return 0;
}
```

Allocate an array of std::string

For further information, check out resources on C++ templates available on the course webpage.