CS 32 - Discussion 1B

Week 6 - Recursion, Templates, & STL

Templates

Templates

Template functions

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

void Swap(string &a, string &b)
{
   string temp;
   temp = a;
   a = b;
   b = temp;
}

void Swap(Fruit &a, Fruit &b)
{
   Fruit temp;
   temp = a;
   a = b;
   b = temp;
}
```

The idea: code reuse

Suppose we write a function which would like to use for many different types.

Templates

Template functions

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

void Swap(string &a, string &b)
{
   string temp;
   temp = a;
   a = b;
   b = temp;
}

void Swap(Fruit &a, Fruit &b)
{
   Fruit temp;
   temp = a;
   a = b;
   b = temp;
}
```

The idea: code reuse

Suppose we write a function which would like to use for many different types.

Writing an implementation for every type seems unnecessary

Templates

Template functions

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

The idea: code reuse

Suppose we write a function which would like to use for many different types.

Writing an implementation for every type seems unnecessary

Fix: use templates to create a “generic” function.

Templates

Template functions

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

int main()
{
    int x = 5;
    int y = 10;
    Swap(5,10); // compiler instantiates code for Swap using int
}
```

Fix: use templates to create a “generic” function.

Templates

Template functions

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

int main()
{
    int x = 5;
    int y = 10;
    Swap(5,10); // compiler instantiates code for Swap using int
    Chicken c1;
    Chicken c2;
    Swap(c1,c2); // compiler instantiates code for Swap using Chicken
}
```

Fix: use templates to create a “generic” function.

Templates

Template functions

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

int main()
{
    int x = 5;
    int y = 10;
    Swap(5,10); // compiler instantiates code for Swap using int
    Chicken c1;
    Chicken c2;
    Swap(c1,c2); // compiler instantiates code for Swap using Chicken
}
```

Template checklist:

1. Call must match some template.

2. Instantiated template must compile.

3. Instantiated template must do what you want it to do.

Templates

Template functions

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

int main()
{
    int x = 5;
    int y = 10;
    Swap(5,10); // compiler instantiates code for Swap using int
    Chicken c1;
    Chicken c2;
    Swap(c1,c2); // compiler instantiates code for Swap using Chicken
}
```

Template checklist:

1. Call must match some template.

2. Instantiated template must compile.

3. Instantiated template must do what you want it to do.

Templates

Template functions

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

int main()
{
    int x = 5;
    int y = 10;
    Swap(5,10); // compiler instantiates code for Swap using int
    Chicken c1;
    Chicken c2;
    Swap(c1,c2); // compiler instantiates code for Swap using Chicken
}
```

Template checklist:

1. Call must match some template.

2. Instantiated template must compile.

3. Instantiated template must do what you want it to do.

If function doesn’t do what you want for a certain type, you can overload it for that type!

Templates

Template functions

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

template <>
void Swap(WeirdType &a, WeirdType &b)
{
  … specific implementation for WeirdType …
}
```

Template checklist:

1. Call must match some template.

2. Instantiated template must compile.

3. Instantiated template must do what you want it to do.

If function doesn't do what you want for a certain type, you can overload it for that type!

Templates

Class templates

```
class intPair
{
  public:
    intPair() : m_x(0), m_y(0) {}
    intPair(int x, int y) : m_x(x), m_y(y) {}
    void display() {cout << “(“ << m_x << “, “ << m_y << “) << endl;}
  private:
    int m_x;
    int m_y;
};

class doublePair
{
  public:
    doublePair() : m_x(0), m_y(0) {}
    doublePair(int x, int y) : m_x(x), m_y(y) {}
    void display() {cout << “(“ << m_x << “, “ << m_y << “)” << endl;}
  private:
    double m_x;
    double m_y;
};
```

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      // default constructor???
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};
```

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
  private:
    Item m_x;
    Item m_y;
};
```

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};

int main() {
   Pair<int> p(10,12); // all good
   Pair<string> p(“beep”,”beep”); // all good
   …
}
```

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};
```

Q. Is there a potential issue?

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};
```

Q. Is there a potential issue?

A. Might error when we run with Item that does not support “cout <<”

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};

// Example : (this compiles and runs!)
int main() {
   Pair<Pair<int>> p;
   return 0;
}
```

Q. Is there a potential issue?

A. Might error when we run with Item that does not support “cout <<”

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};

// Example : (This does not run)
int main() {
   Pair<Pair<int>> p;
   p.display(); // throws error
   return 0;
}
```

Q. Is there a potential issue?

A. Might error when we run with Item that does not support “cout <<”

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << “(” << m_x << “,” << m_y << “)” << endl;}
   private:
      Item m_x;
      Item m_y;
};

// Example : (This does not run)
int main() {
   Pair<Pair<int>> p;
   p.display(); // throws error
   return 0;
}
```

A possible fix using a **template function**:

using **to_string**

these are all built-in in C++:

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

- converts input to a string in the expected way

Templates

Class templates

```
template <typename Item>
class Pair
{
   public:
      Pair() : m_x(Item()), m_y(Item()) {}
      Pair(Item x, Item y) : m_x(x), m_y(y) {}
      void display() const {cout << to_string(*this) << endl;}
   private:
      Item m_x;
      Item m_y;
};
```

A possible fix using a **template function**:

using **to_string**

these are all built-in in C++:

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

- converts input to a string in the expected way

Idea: write a template to_string function for Pairs

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
```

```
template <typename Item>
string to_string(const Pair<Item>& p) {
  return “(” + to_string(p.getx()) + “,” + to_string(p.gety()) + “)”;
}
```

A possible fix: using to_string

these are all built-in in C++

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

- converts input to a string in the expected way

Idea: write a template to_string function for Pairs

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
// Works as long as Item has a to_string function implemented!
template <typename Item>
string to_string(const Pair<Item>& p) {
  return “(” + to_string(p.getx()) + “,” + to_string(p.gety()) + “)”;
}
```

A possible fix: using to_string

these are all built-in in C++

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

- converts input to a string in the expected way

Idea: write a template to_string function for Pairs

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
// Works as long as Item has a to_string function implemented!
template <typename Item>
string to_string(const Pair<Item>& p) {
  return "(" + to_string(p.getx()) + "," + to_string(p.gety()) + ")";
}
```

Making display work using to_string

```
// Now this works!
int main() {
  Pair<Pair<int>> p(Pair<int>(1,2),Pair<int>(3,4));
  p.display(); // prints "((1,2),(3,4))"
  return 0;
}
```

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
// Note: this function is recursive… ?
template <typename Item>
string to_string(const Pair<Item>& p) {
  return “(” + to_string(p.getx()) + “,” + to_string(p.gety()) + “)”;
}
```

Making display work using to_string

```
// Now this works!
int main() {
  Pair<Pair<int>> p(Pair<int>(1,2),Pair<int>(3,4));
  p.display(); // prints “((1,2),(3,4))”
  return 0;
}
```

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
// Note: this function is recursive… but there's no base case … ??
template <typename Item>
string to_string(const Pair<Item>& p) {
  return "(" + to_string(p.getx()) + "," + to_string(p.gety()) + ")";
}
```

Making display work using to_string

```
// Now this works!
int main() {
  Pair<Pair<int>> p(Pair<int>(1,2),Pair<int>(3,4));
  p.display(); // prints "((1,2),(3,4))"
  return 0;
}
```

Templates

Class templates

```
template <typename Item>
class Pair
{
  public:
    Pair() : m_x(Item()), m_y(Item()) {}
    Pair(Item x, Item y) : m_x(x), m_y(y) {}
    Item getx() const {return m_x};
    Item gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item m_x;
    Item m_y;
};
// Note: this function is recursive… but there's no base case … ??
// base case is implicit: eventually will be calling to_string on a primitive type…
// to_string for primitives is not recursive.
template <typename Item>
string to_string(const Pair<Item>& p) {
  return "(" + to_string(p.getx()) + "," + to_string(p.gety()) + ")";
}
```

Making display work using to_string

```
// Now this works!
int main() {
  Pair<Pair<int>> p(Pair<int>(1,2),Pair<int>(3,4));
  p.display(); // prints "((1,2),(3,4))"
  return 0;
}
```

Templates

Class templates

Extending to multiple types…

```
template <typename Item1, typename Item2>
class Pair
{
  public:
    Pair() : m_x(Item1()), m_y(Item2()) {}
    Pair(Item1 x, Item2 y) : m_x(x), m_y(y) {}
    Item1 getx() const {return m_x};
    Item2 gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item1 m_x;
    Item2 m_y;
};

template <typename Item1, typename Item2>
string to_string(const Pair<Item1, Item2>& p) {
  return “(” + to_string(p.getx()) + “,” + to_string(p.gety()) + “)”;
}
```

Templates

Class templates

```
template <typename Item1, typename Item2>
class Pair
{
  public:
    Pair() : m_x(Item1()), m_y(Item2()) {}
    Pair(Item1 x, Item2 y) : m_x(x), m_y(y) {}
    Item1 getx() const {return m_x};
    Item2 gety() const {return m_y};
    void display() const {cout << to_string(*this) << endl;}
  private:
    Item1 m_x;
    Item2 m_y;
};

template <typename Item1, typename Item2>
string to_string(const Pair<Item1, Item2>& p) {
  return “(” + to_string(p.getx()) + “,” + to_string(p.gety()) + “)”;
}

string to_string(const string& s) {return s;}
```

Extending to multiple types…

```
// Now this works!
int main() {
  Pair<double,Pair<int,string>> p1;
  Pair<double,Pair<int,string>> p2(3.1415, Pair<int,string>(200, “beep beep beeeeep”));

  p2.display(); // prints: “(3.141500,(200,beep beep beeeeep))”
  return 0;
}
```

Extremely quick review

STL
(Standard Template Library)

Extremely quick review

STL
(Standard Template Library)

A collection of pre-written, tested classes and algorithms built using templates

Extremely quick review

STL
(Standard Template Library)

A collection of pre-written, tested classes and algorithms built using templates

=> Flexibility to use with many different data types

STL

STL container classes

#include <stack> // already seen this
#include <queue> // already seen this
#include <vector> // implemented using an array
#include <list> // implemented using doubly linked list

Vector: Basically an **automatically resizing** array with some built-in functions.
vector<int> vi; // note the template syntax because its a template class!
List: A doubly linked list with some built-in functions.
list<int> li;

STL

STL iterators

Iterators give a general way of iterating through the elements of a container class

Each container class has a iterator

implemented differently for different container classes

… but uses the exact same syntax for each.

Example:

```
vector<int> v = {1,2,3,4,5};
for(vector<int>::iterator p = v.begin(); p != v.end(); p++)
{
   cout << *p << endl;
}
```

Reference for info on STL Container Classes

https://en.cppreference.com/w/cpp/container

- Lists and describes supported functions
  - + example code
- Describes how each container class is implemented
- Describes complexity of each supported operation