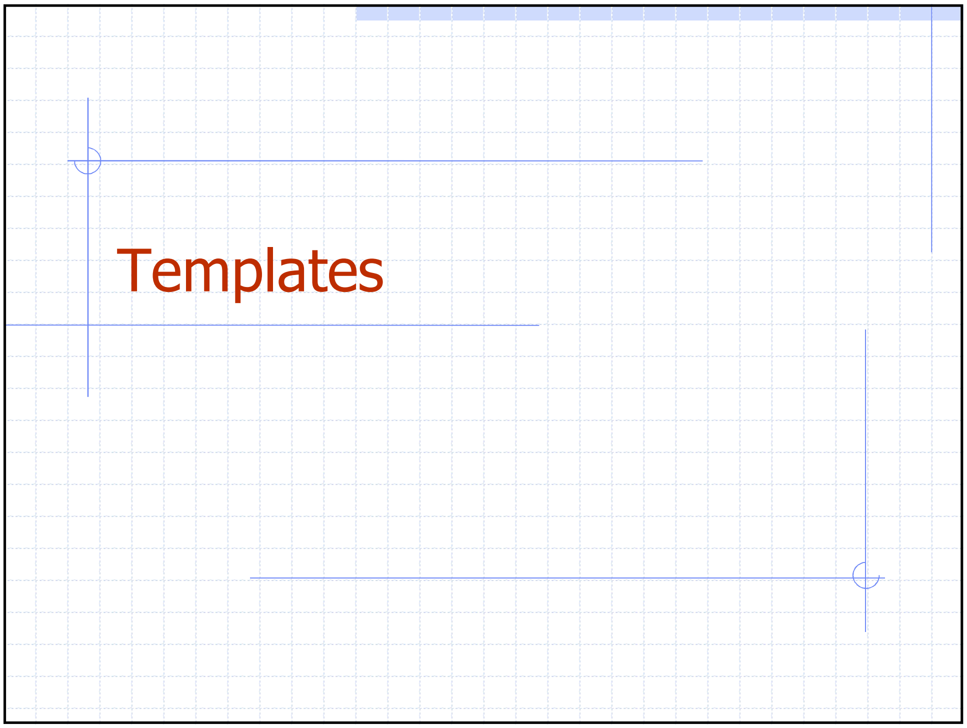
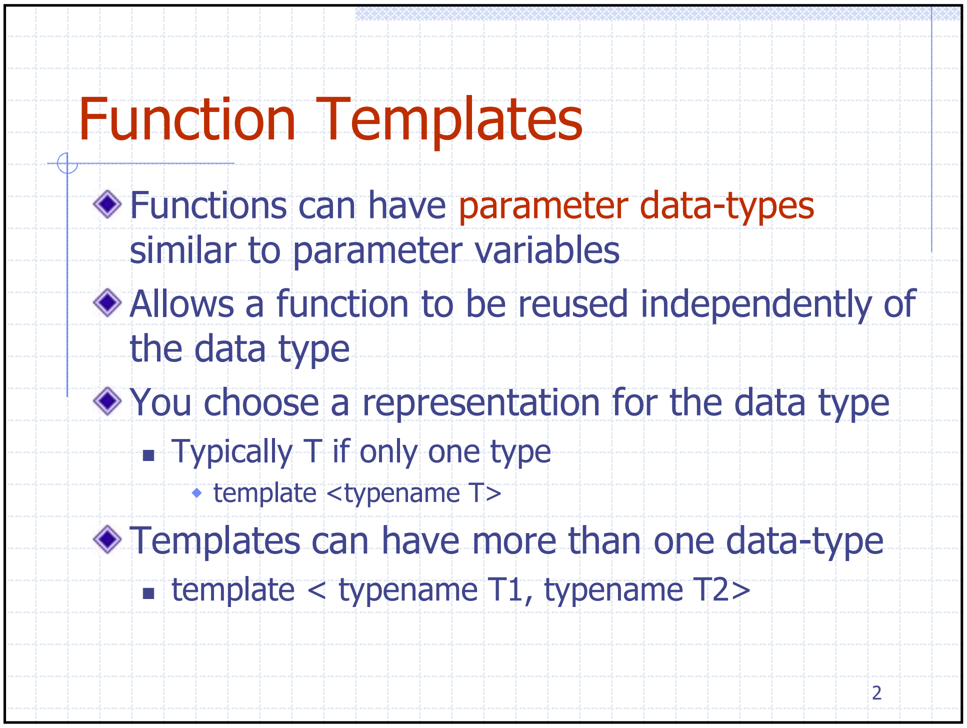
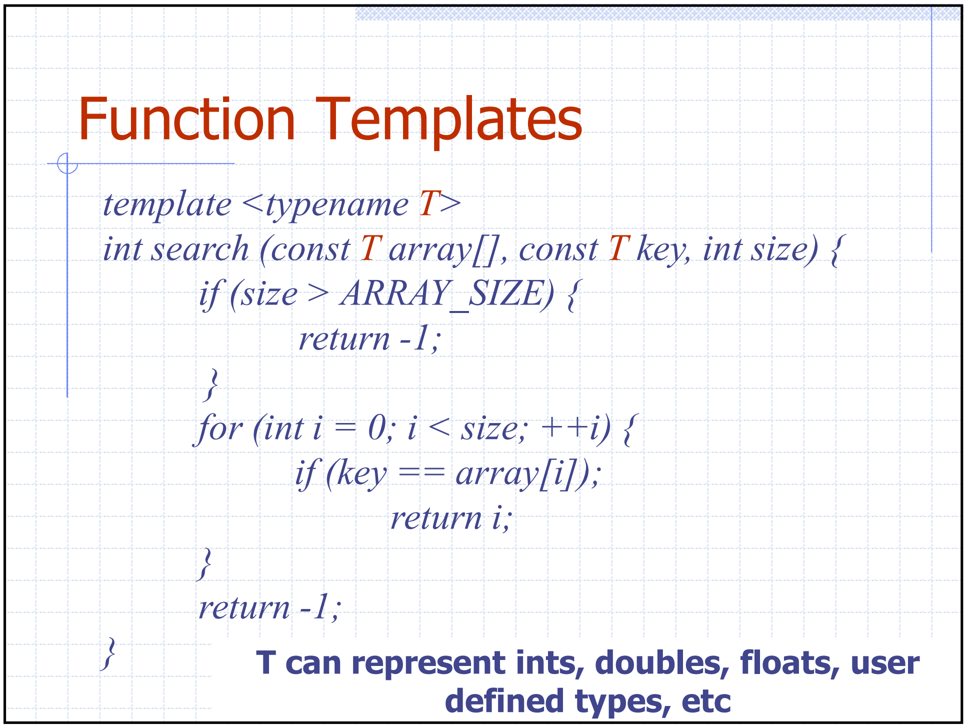


Templates



Function Templates

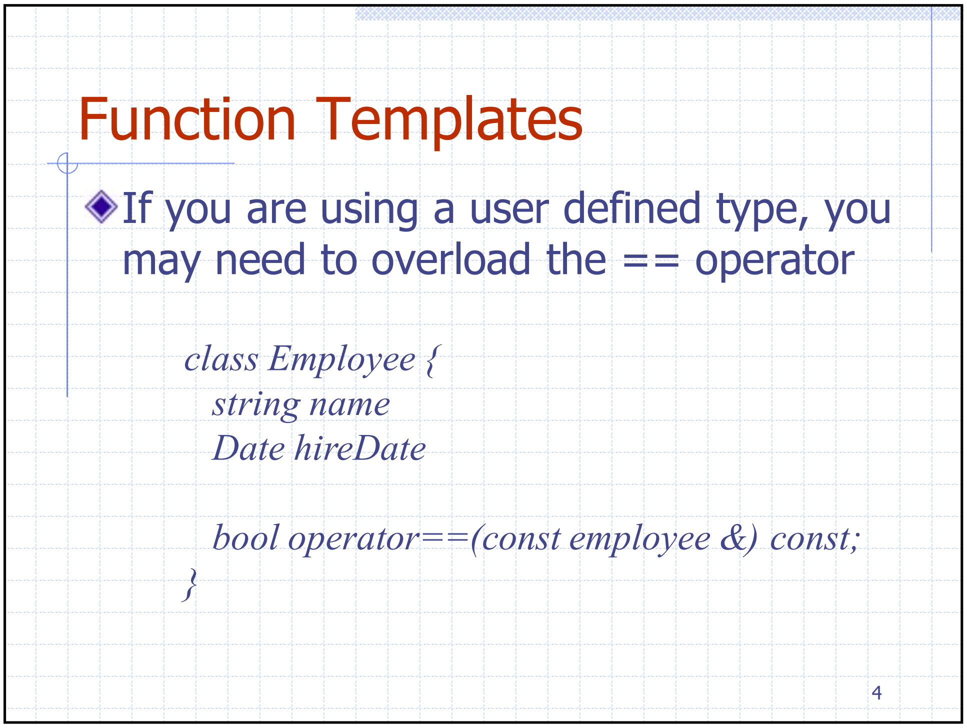
- Functions can have parameter data-types similar to parameter variables
- Allows a function to be reused independently of the data type
- You choose a representation for the data type
  - Typically T if only one type
    - template <typename T>
- Templates can have more than one data-type
  - template < typename T1, typename T2>



Function Templates

```
template <typename T>
int search (const T array[], const T key, int size) {
        if (size > ARRAY_SIZE) {
                return -1;
        }
        for (int i = 0; i < size; ++i) {
                if (key == array[i]);
                        return i;
        }
        return -1;
}
```

T can represent ints, doubles, floats, user defined types, etc

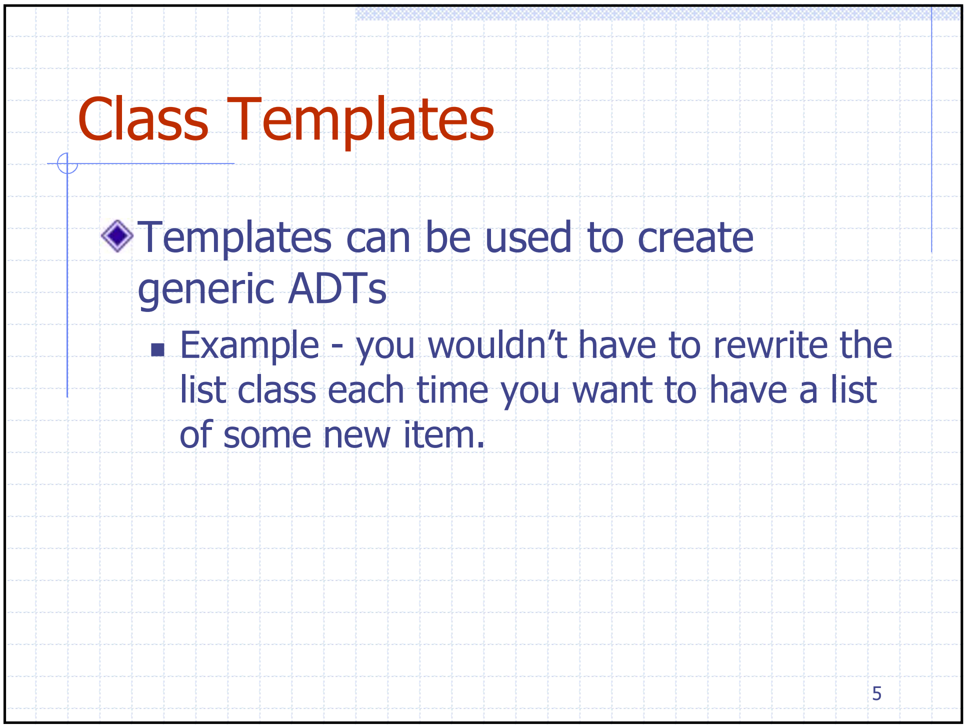


Function Templates

- If you are using a user defined type, you may need to overload the == operator

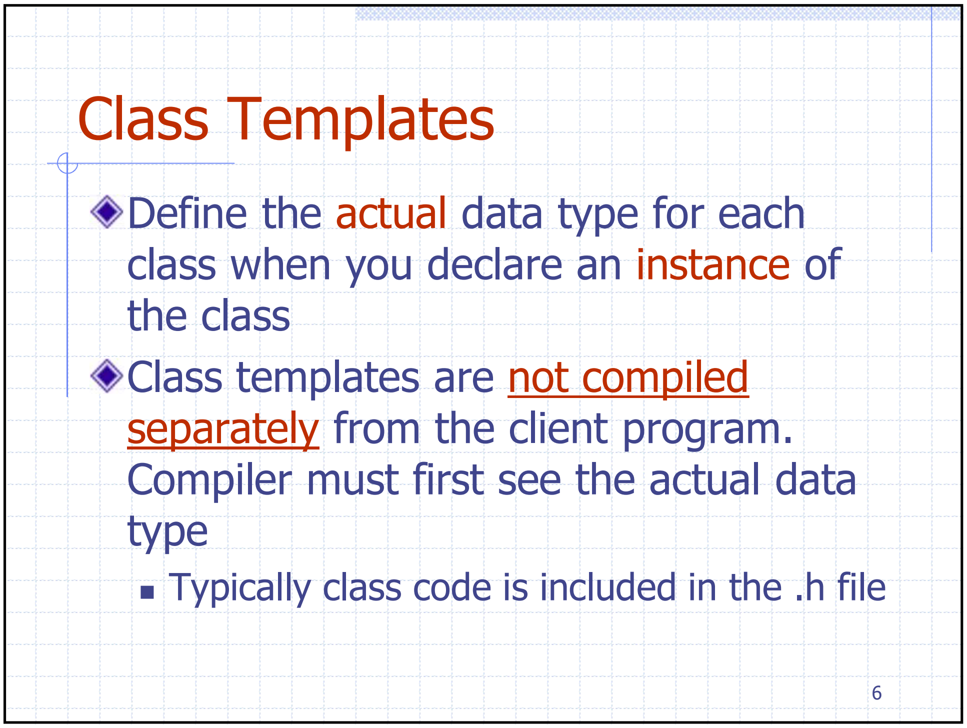
```
class Employee {
  string name
  Date hireDate

  bool operator==(const employee &) const;
}
```



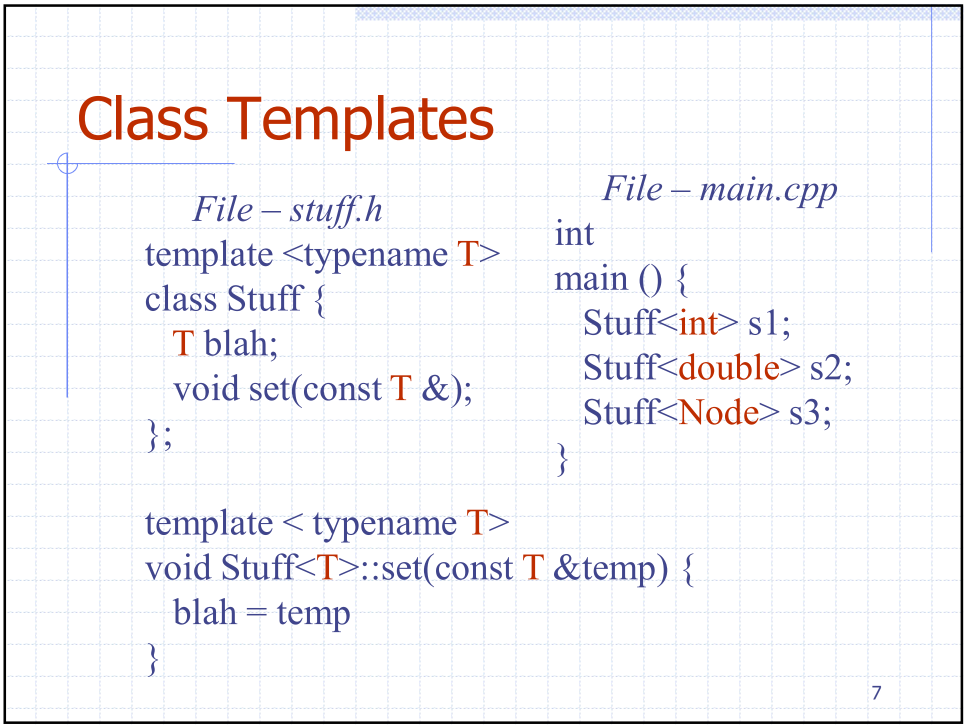
Class Templates

- Templates can be used to create generic ADTs
  - Example - you wouldn’t have to rewrite the list class each time you want to have a list of some new item.



Class Templates

- Define the actual data type for each class when you declare an instance of the class
- Class templates are not compiled separately from the client program. Compiler must first see the actual data type
  - Typically class code is included in the .h file



Class Templates

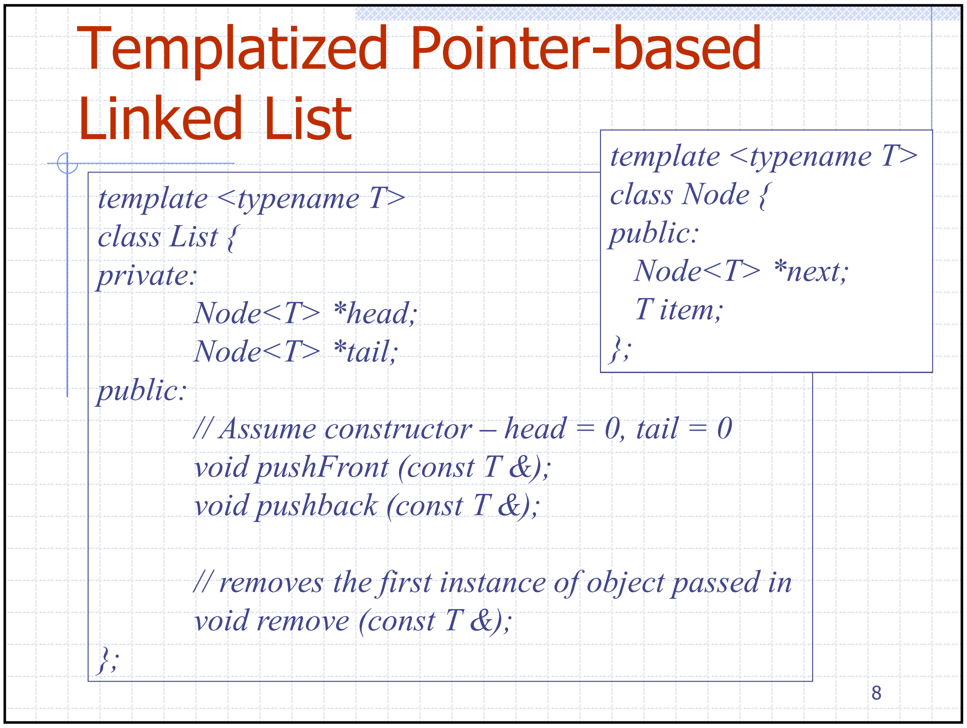
File – stuff.h

```
template <typename T>
class Stuff {
   T blah;
   void set(const T &);
};
```

File – main.cpp

```
int
main () {
   Stuff<int> s1;
   Stuff<double> s2;
   Stuff<Node> s3;
}
```

```
template < typename T>
void Stuff<T>::set(const T &temp) {
   blah = temp
}
```

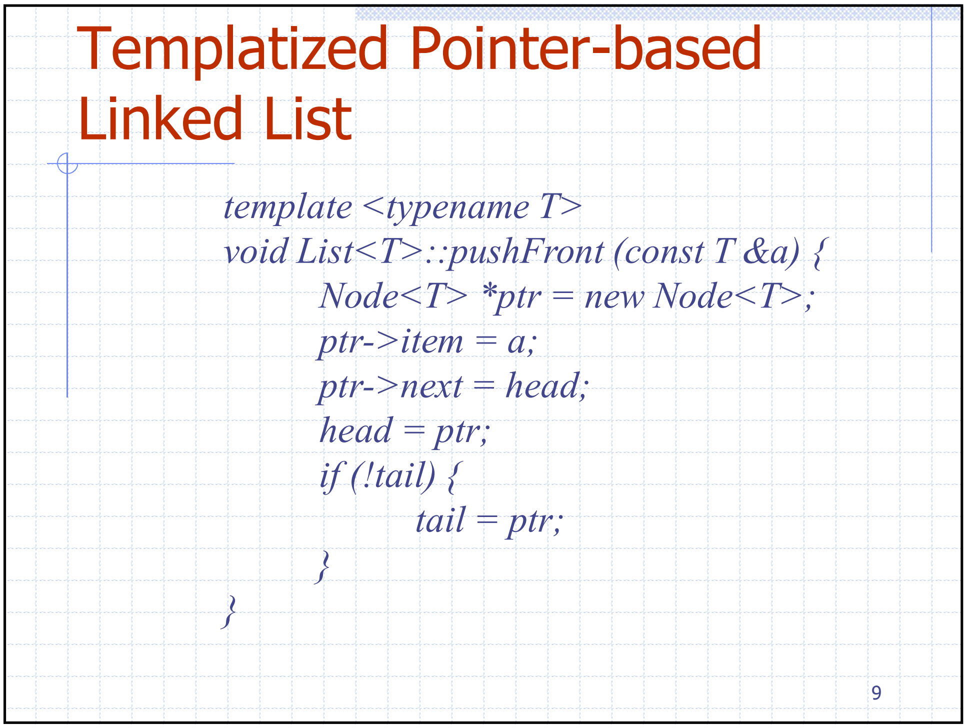


Templatized Pointer-based Linked List

```
template <typename T>
class Node {
public:
   Node<T> *next;
   T item;
};
```

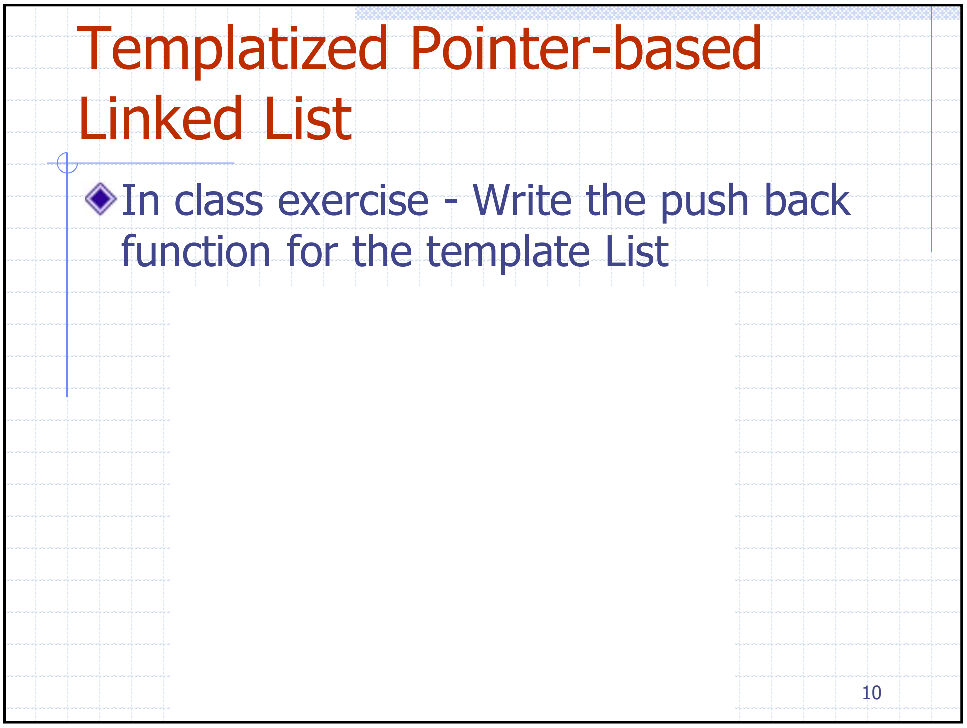
```
template <typename T>
class List {
private:
          Node<T> *head;
          Node<T> *tail;
public:
          // Assume constructor – head = 0, tail = 0
          void pushFront (const T &);
          void pushback (const T &);

          // removes the first instance of object passed in
          void remove (const T &);
};
```



Templatized Pointer-based Linked List

```
template <typename T>
void List<T>::pushFront (const T &a) {
        Node<T> *ptr = new Node<T>;
        ptr->item = a;
        ptr->next = head;
        head = ptr;
        if (!tail) {
                tail = ptr;
        }
}
```



Templatized Pointer-based Linked List

- In class exercise - Write the push back function for the template List