# Templates and Generics

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: February 6, 2025

# Parametric polymorphism: template functions, template classes

# Motivation

- Want to define both uniformly

# Motivation

- Want to define both uniformly

# Motivation

- Want to define both uniformly

```
int MaxInt(int a, int b) {
  if (b < a)
    return a;
  else
    return b;
}

double MaxDouble(double a, double b) {
  if (b < a)
    return a;
  else
    return b;
}
```

# Idea

- Can define a generic function with generic parameters

# Idea

- Can define a generic function with generic parameters

# Idea

- Can define a generic function with generic parameters

```
SomeType MaxSomeType(SomeType a, SomeType b) {
  if (b < a)
    return a;
  else
    return b;
}
```

- What properties does `SomeType` need for this to work?

# Implementing using Templates

- Keywords: `template`, `typename`

# Implementing using Templates

- Keywords: `template`, `typename`

# Implementing using Templates

- Keywords: `template`, `typename`

```
// can also use the keyword class rather than typename
template <typename T>
T Max(T a, T b) {
  if (b < a)
    return a;
  else
    return b;
}

Max<int>(3, 4); // or in most cases, simply: Max(3, 4);
```

# Program organization

- Where should we place function templates?

# Program organization

- Where should we place function templates?
    - inclusion compilation model vs. separate compilation model

# Program organization

- Where should we place function templates?
    - inclusion compilation model vs. separate compilation model
- We will use inclusion compilation model (as it is supported by all compilers)

# Program organization

- Where should we place function templates?
    - inclusion compilation model vs. separate compilation model
- We will use inclusion compilation model (as it is supported by all compilers)
- Idea:

# Program organization

- Where should we place function templates?
    - inclusion compilation model vs. separate compilation model
- We will use inclusion compilation model (as it is supported by all compilers)
- Idea:
    - place template in a header file

# Program organization

- Where should we place function templates?
  - inclusion compilation model vs. separate compilation model
- We will use inclusion compilation model (as it is supported by all compilers)
- Idea:
  - place template in a header file
  - the compiler will only generate code on instantiation

# Program organization

- Where should we place function templates?
    - inclusion compilation model vs. separate compilation model
- We will use inclusion compilation model (as it is supported by all compilers)
- Idea:
    - place template in a header file
    - the compiler will only generate code on instantiation
        - avoids “code bloat” suffered by early implementations

# Restrictions on template abstraction

- Only works for types which have the necessary operations defined

# Restrictions on template abstraction

- Only works for types which have the necessary operations defined

# Restrictions on template abstraction

- Only works for types which have the necessary operations defined

```
// works since string class overloads <
Max(string("abc"), string("def"));

Max("abc", "def"); // WRONG, as < is not defined for C strings
// i.e. arrays of characters
```

# Restrictions ctd.

- Similarly, won't work for other types that do not define <

# Restrictions ctd.

- Similarly, won’t work for other types that do not define <

# Restrictions ctd.

- Similarly, won’t work for other types that do not define <

```
struct Book {
  string author;
  string title;
};

Book b1, b2;
b1.author = "Me";
b1.title = "BestSeller";
b2.author="You";
b2.title= "Whatever!";

Max(b1,b2); // WRONG!
```

# Making The Example Work

- It will compile if we add the following:

# Making The Example Work

- It will compile if we add the following:

# Making The Example Work

- It will compile if we add the following:

```cpp
bool operator<(const Book &b1, const Book &b2) {
  return (b1.author < b2.author)
         ((b1.author == b2.author) && (b1.title < b2.title));
```

- Only then:

# Making The Example Work

- It will compile if we add the following:

```cpp
bool operator<(const Book &b1, const Book &b2) {
  return (b1.author < b2.author)
         ((b1.author == b2.author) && (b1.title < b2.title));
```

- Only then:

# Making The Example Work

- It will compile if we add the following:

```
bool operator<(const Book &b1, const Book &b2) {
  return (b1.author < b2.author)
         ((b1.author == b2.author) && (b1.title < b2.title));
```

- Only then:

```
Max(b1,b2); // Works!
```

# Specifying template abstraction

# Specifying template abstraction

```
// Max.h
//
#pragma once
//
// Max<T>(a, b)
// Purpose: Find the maximum of two given arguments.
// Template Parameter(s):
//  <1> T: A type for which the following operations are defined:
//    -> copy constructor
//       [usually automatically created by C++ compilers]
//    -> binary less than comparison (<)
// Parameter(s):
//  <1> a: An instances of type T
//  <2> b: An instances of type T
// Precondition(s): N/A
// Returns: A T-type value equivalent to the maximum of a and b.
// Side Effect: N/A
```

## Reducing the Requirements

- Earlier, could have dropped the copy constructor requirement by passing references instead:

# Reducing the Requirements

- Earlier, could have dropped the copy constructor requirement by passing references instead:

# Reducing the Requirements

- Earlier, could have dropped the copy constructor requirement by passing references instead:

```cpp
template <typename T>
T &Max(T &a, T &b) {
  if (b < a)
    return a;
  else
    return b;
}
```

- Better implementation as doesn’t waste memory by creating temporary objects

# Selection Sort

- Recall, we had `int` hard-coded in

# Selection Sort

- Recall, we had `int` hard-coded in

## Selection Sort

- Recall, we had `int` hard-coded in

```
typedef int ItemType;

void Swap(ItemType &a, ItemType &b){
  ItemType tmp = a;
  a = b;
  b = tmp;
}
unsigned int FindMin(const ItemType A[],
                     unsigned int begin,
                     unsigned int end){
  ...
    }
void Sort(ItemType A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = FindMin(A, i, n - 1);
    Swap(A[i], A[m]);
  }
}
```

# Generic Helper Functions

- Should work for anything with <

# Generic Helper Functions

- Should work for anything with <

# Generic Helper Functions

- Should work for anything with <

```
template <typename ItemType>
void Swap(ItemType &a, ItemType &b){
  ItemType tmp = a;        // copy constructor
  a = b;                           // assignment operator
  b = tmp;

}
template <typename ItemType>
unsigned int FindMin(const ItemType A[],
                     unsigned int begin,
                     unsigned int end){
  assert(begin <= end);
  unsigned int m = begin;
  for (unsigned int i = begin + 1; i <= end; i++){
    if (A[i] < A[m])    // less than comparison operator
      m = i;
  }
  return m;
}
```

# A Fully Generic Sort

# A Fully Generic Sort

```
template <typename ItemType>
void Sort(ItemType A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = FindMin(A, i, n - 1);
    Swap(A[i], A[m]);
  }
}
```

- Thus the interface should include the following requirements:

# A Fully Generic Sort

```
template <typename ItemType>
void Sort(ItemType A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = FindMin(A, i, n - 1);
    Swap(A[i], A[m]);
  }
}
```

- Thus the interface should include the following requirements:

# A Fully Generic Sort

```
template <typename ItemType>
void Sort(ItemType A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = FindMin(A, i, n - 1);
    Swap(A[i], A[m]);
  }
}
```

- Thus the interface should include the following requirements:

```
// Template Parameter(s):
//  <1> ItemType: A type for which
//          the following operations are defined:
//    -> copy constructor
//    -> assignment operator
//    -> binary less than comparison (<)
```

# Multiple Typenames

- Can have multiple different type names in template arguments (separated by a comma), e.g.:

# Multiple Typenames

- Can have multiple different type names in template arguments (separated by a comma), e.g.:

# Multiple Typenames

- Can have multiple different type names in template arguments (separated by a comma), e.g.:

```
template <typename T, typename K, typename O>
T func1(K a, O b) {
  T x, y;
  ...
    if (func2(a,b)==x)
    return x;
  else
    return y;
}
```

# Functions as Template Parameters

- Could have generalized our selection sort template to sort either in ascending or in descending order by replacing < with a function

# Functions as Template Parameters

- Could have generalized our selection sort template to sort either in ascending or in descending order by replacing < with a function

# Functions as Template Parameters

- Could have generalized our selection sort template to sort either in ascending or in descending order by replacing < with a function

```cpp
template <typename T, bool compare(const T &x, const T &y)>
unsigned int Find(const T A[],
                  unsigned int begin,
                  unsigned int end) {
  assert(begin <= end);
  unsigned int m = begin;
  for (unsigned int i = begin + 1; i <= end; i++){
    if (compare(A[i], A[m]))
      m = i;
  }
  return m;
}
```

# Function Parameters ctd.

- New template for sort

# Function Parameters ctd.

- New template for sort

# Function Parameters ctd.

- New template for sort

```
template <typename T, bool compare(const T &x, const T &y)>
void Sort(T A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = Find<T, compare>(A, i, n - 1);
    Swap(A[i], A[m]);
  }
}
```

# Template Client Code

- Can explicitly specify which operation to fill the template in with

# Template Client Code

- Can explicitly specify which operation to fill the template in with

# Template Client Code

- Can explicitly specify which operation to fill the template in with

```
bool less_than(const int &x, const int &y){
  return x < y;
}
...
Sort<int, less_than>(...);

bool greater_than(const int &x, const int &y){
  return x > y;
}
...
Sort<int, greater_than>(...);
```

# Template classes

- Can also define classes in terms of type variables

# Template classes

- Can also define classes in terms of type variables
  - Useful for defining *containers*

# Template classes

- Can also define classes in terms of type variables
    - Useful for defining *containers*
    - Want to specify *how* the data is structured, but not what type it is

# Template classes

- Can also define classes in terms of type variables
    - Useful for defining *containers*
    - Want to specify *how* the data is structured, but not what type it is
- E.g. Guarded arrays

# Template classes

- Can also define classes in terms of type variables
  - Useful for defining *containers*
  - Want to specify *how* the data is structured, but not what type it is
- E.g. Guarded arrays
  - Just like before, but not restricted to integer elements

# Template classes

- Can also define classes in terms of type variables
  - Useful for defining *containers*
  - Want to specify *how* the data is structured, but not what type it is
- E.g. Guarded arrays
  - Just like before, but not restricted to integer elements

# Template classes

- Can also define classes in terms of type variables
    - Useful for defining *containers*
    - Want to specify *how* the data is structured, but not what type it is
- E.g. Guarded arrays
    - Just like before, but not restricted to integer elements

```
template<typename T>

class GuardedArray {
public:
  static const unsigned int LENGTH = 500;
  GuardedArray();
  GuardedArray(T x);
  ItemType retrieve(unsigned int i) const;
  void store(unsigned int i, T x);
private:
  T data_array[LENGTH];
};
```