CSC 211: Computer Programming

Templating, size_t

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Fall 2024

THINK BIG WE DO℠

Templating

Templating

- Template programming in C++ is a powerful feature that allows you to write generic code that works with different data types without sacrificing type safety

- Templates enable you to define functions and classes with generic types, and the compiler generates specific instances of the code for each type used.

- This results in more flexible and reusable code.

Types of Templates

- Function Templates
  - ✓ Allow you to write a single function that can operate on different data types

- Class Templates
  - ✓ Allow you to create generic classes that can work with different data types

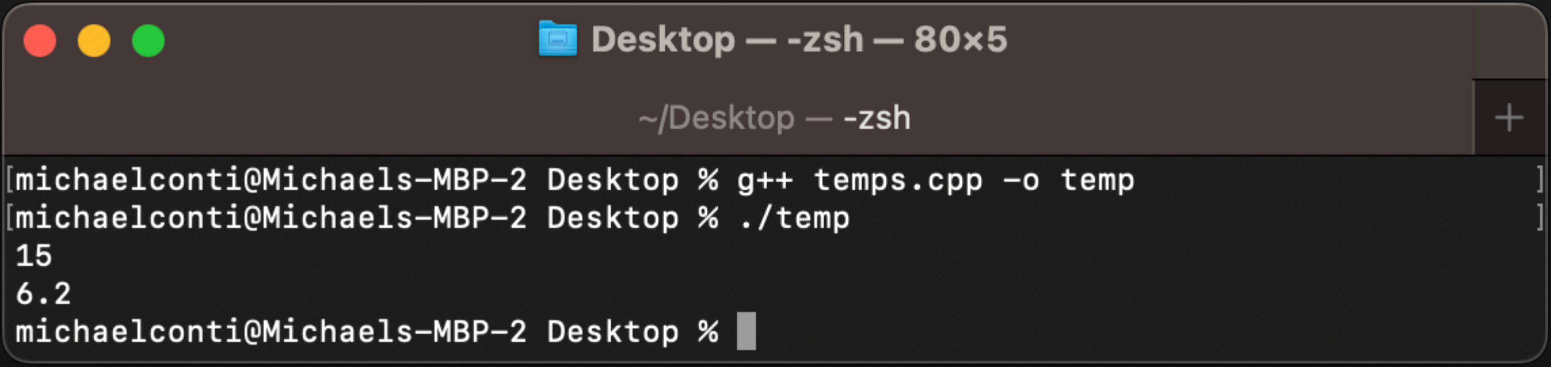
- Uses the reserved keyword **template** followed parameter **typename** or **class**

Syntax

```
template <typename identifier>
```

Function Templates

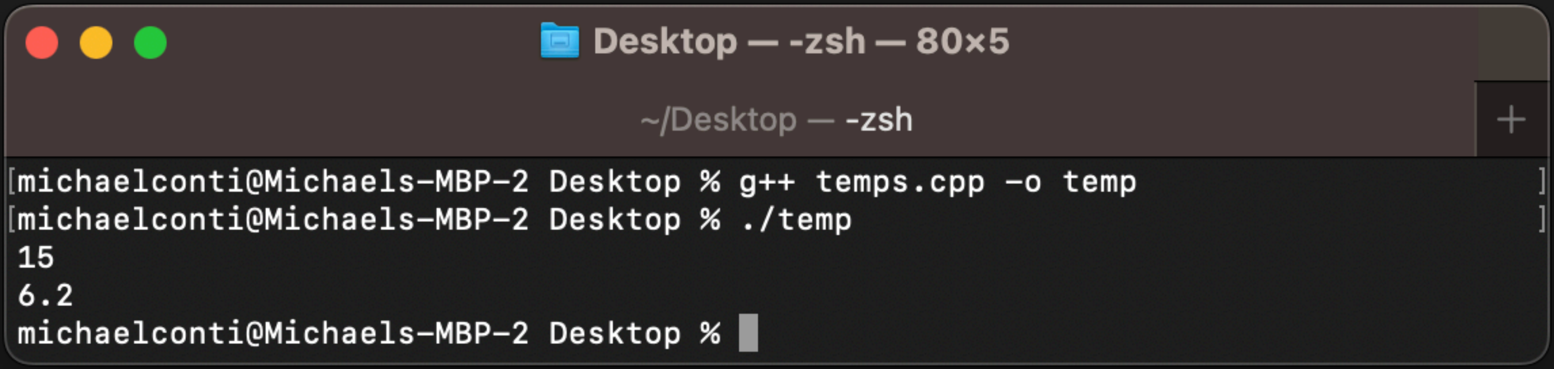
```
template <typename T>

T add(T a, T b) {
    return a + b;
}

int main() {
    int result1 = add(5, 10);       // Calls add<int>(5, 10)
    double result2 = add(3.5, 2.7);  // Calls add<double>(3.5, 2.6)

    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;
```



```
[michaelconti@Michaels-MBP-2 Desktop % g++ temps.cpp -o temp                    ]
[michaelconti@Michaels-MBP-2 Desktop % ./temp                                   ]
15
6.2
michaelconti@Michaels-MBP-2 Desktop % 
```

Function Templates

```cpp
template <typename pizza>

pizza add(pizza a, pizza b) {
    return a + b;
}

int main() {
    int result1 = add(5, 10);          // Calls add<int>(5, 10)
    double result2 = add(3.5, 2.7);  // Calls add<double>(3.5, 2.5)

    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;
```



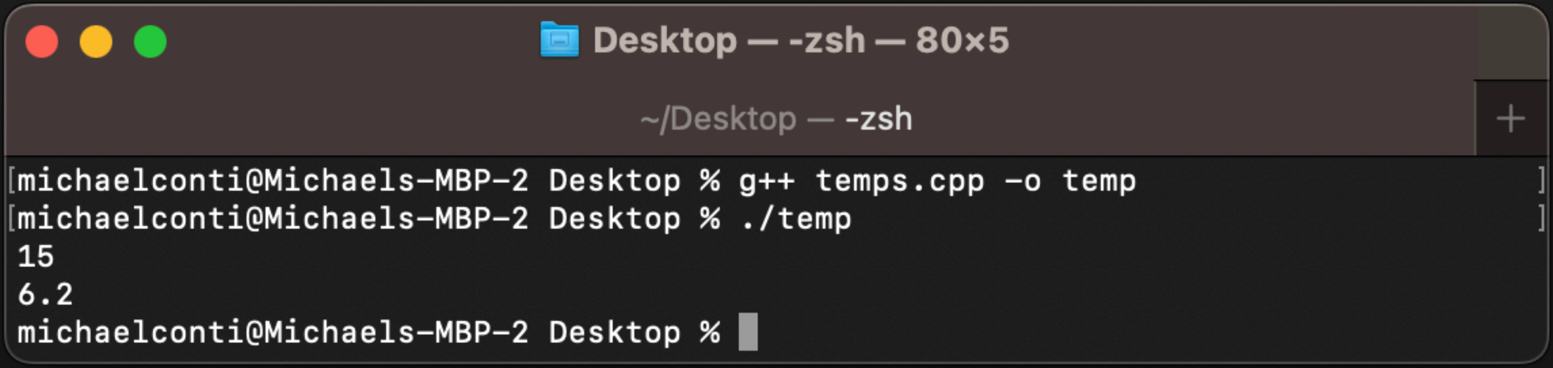
```
[michaelconti@Michaels-MBP-2 Desktop % g++ temps.cpp -o temp                    ]
[michaelconti@Michaels-MBP-2 Desktop % ./temp                                   ]
15
6.2
michaelconti@Michaels-MBP-2 Desktop % 
```

Function Templates

```
template <class T>
```

Can also use ‘class’ instead of typename

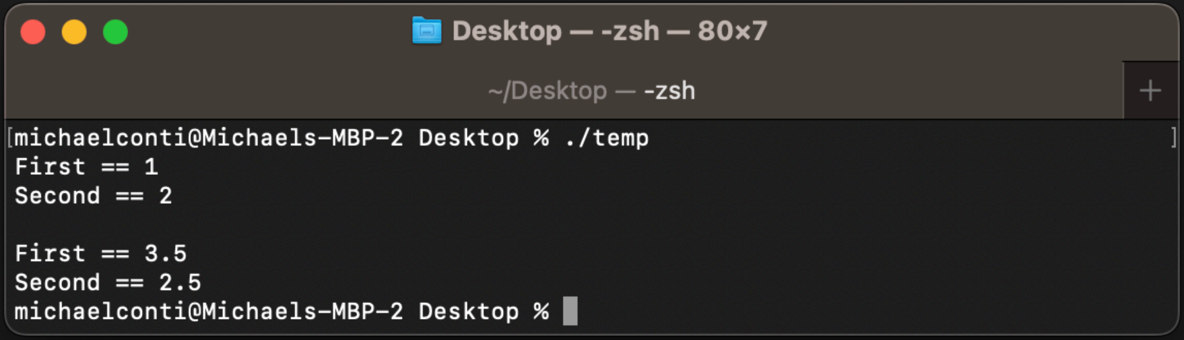
```
T add(T a, T b) {
    return a + b;
}

int main() {
    int result1 = add(5, 10);          // Calls add<int>(5, 10)
    double result2 = add(3.5, 2.7);  // Calls add<double>(3.5, 2.6)

    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;
```



```
[michaelconti@Michaels-MBP-2 Desktop % g++ temps.cpp -o temp                    ]
[michaelconti@Michaels-MBP-2 Desktop % ./temp                                    ]
15
6.2
michaelconti@Michaels-MBP-2 Desktop % █
```

Class Templates

```
template <typename T>

class Pair {
    public:
        T first;
        T second;

        Pair(T a, T b) : first(a), second(b) {}

        void print(){
            std::cout << "First == " << first << std::endl;
            std::cout << "Second == " << second << std::endl;
        }
};

int main() {
    Pair<int> intPair(1, 2);
    Pair<double> doublePair(3.5, 2.5);

    intPair.print();

    std::cout << std::endl;

    doublePair.print();

    return 0;
}
```



```
michaelconti@Michaels-MBP-2 Desktop % ./temp
First == 1
Second == 2

First == 3.5
Second == 2.5
michaelconti@Michaels-MBP-2 Desktop % 
```

Template Arguments

- A template argument refers to the type, value, or template itself that is supplied to a template when it is used to create a specific instance of a function template or class template.

- Passed at object creation

Template Arguments

```cpp
// Function template with type template arguments
template<typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Explicitly specifying template arguments
    int sum = add<int>(5, 10);

    // Implicitly deducing template arguments
    float result = add(3.5f, 2.7f);

    return 0;
}
```

Template Specialization

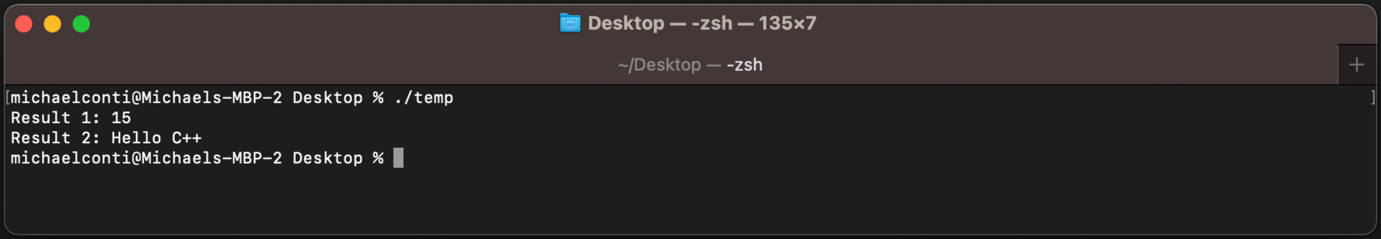
- Template specialization allows you to provide a **custom implementation** for a specific set of template arguments

- **Tailor the behavior** of a template for particular data types or configurations

- Template specialization is particularly useful when the default behavior of a template is **not suitable** for certain types or situations

Function Template Specialization

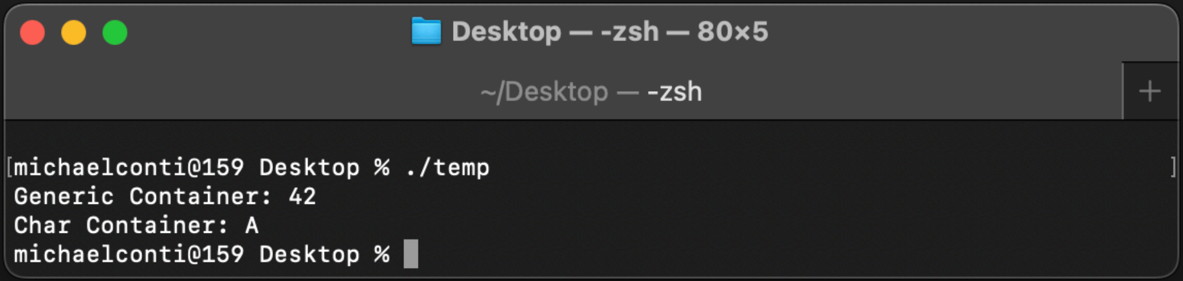
```cpp
// Generic template function
template <typename T>
T add(T a, T b) {
    return a + b;
}

// Template specialization for strings
template <>
std::string add(std::string a, std::string b) {
    return a + " " + b;
}

int main() {
    int result1 = add(5, 10);// Calls add<int>(5, 10)

    std::string str1 = "Hello";
    std::string str2 = "C++";

    std::string result2 = add(str1, str2); // Calls specialized add<std::string>("Hello", "C++")

    std::cout << "Result 1: " << result1 << std::endl;
    std::cout << "Result 2: " << result2 << std::endl;

    return 0;
}
```



```
michaelconti@Michaels-MBP-2 Desktop % ./temp
Result 1: 15
Result 2: Hello C++
michaelconti@Michaels-MBP-2 Desktop % 
```

Class Template Specialization

```cpp
#include <iostream>

// Generic template class
template <typename T>
class Container {
public:
    Container(T value) : data(value) {} // Constructor call with initializer list

    void print() {
        std::cout << "Generic Container: " << data << std::endl;
    }

private:
    T data;
};

// Template specialization for char type
template <>
class Container<char> {
public:
    Container(char value) : data(value) {}

    void print() {
        std::cout << "Char Container: " << data << std::endl;
    }

private:
    char data;
};

int main() {
    Container<int> genericContainer(42);
    Container<char> charContainer('A');

    genericContainer.print(); // Outputs: Generic Container: 42
    charContainer.print();    // Outputs: Char Container: A

    return 0;
}
```



```
michaelconti@159 Desktop % ./temp
Generic Container: 42
Char Container: A
michaelconti@159 Desktop %
```

size_t

size_t

- **size_t** is an unsigned integral (int) type, stands for **"size type"**

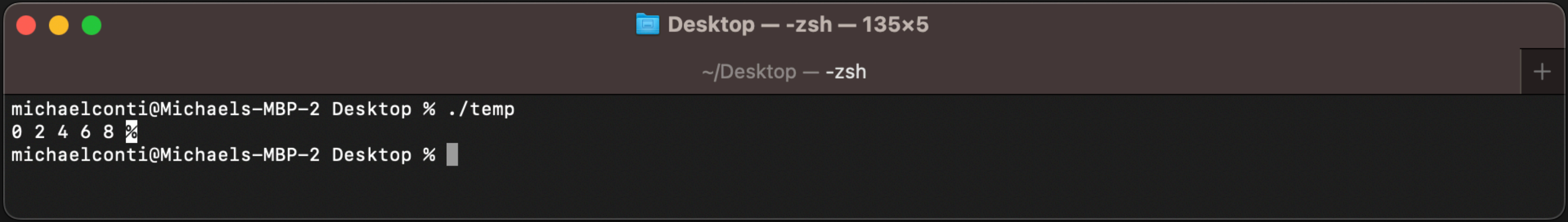
- Commonly used to represent sizes and indices, especially in the context of memory-related operations

- An implementation-specific unsigned integer type and is typically used to ensure portability across different systems.

- "**Basically** an int datatype"

Usage in Array indicies

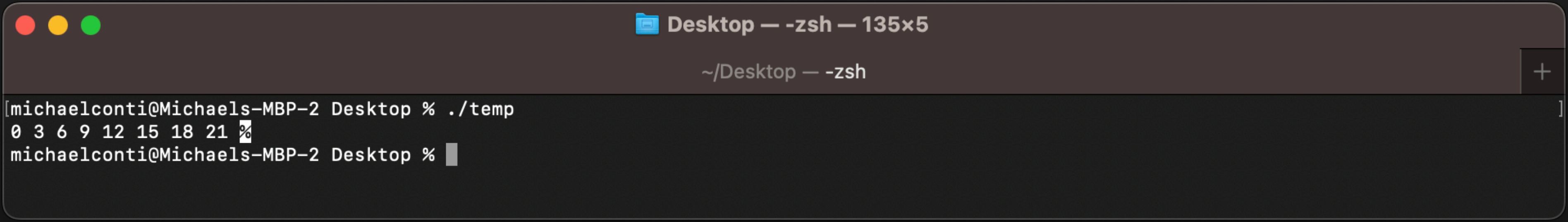
```cpp
int main() {
    const size_t arraySize = 5;
    int myArray[arraySize];

    for (size_t i = 0; i < arraySize; ++i) {
        myArray[i] = i * 2;
        std::cout << myArray[i] << " ";
    }


    return 0;
}
```



```
michaelconti@Michaels-MBP-2 Desktop % ./temp
0 2 4 6 8 %
michaelconti@Michaels-MBP-2 Desktop %
```

Usage in Container Sizes

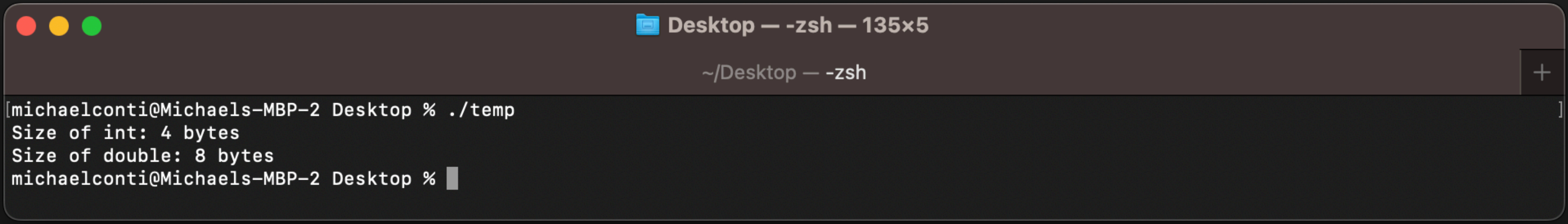
```
int main() {
    std::vector<int> myVector;
    const size_t vectorSize = 8;

    for (size_t i = 0; i < vectorSize; ++i) {
        myVector.push_back(i * 3);

        std::cout << myVector[i] << " ";
    }

    return 0;
}
```



```
[michaelconti@Michaels-MBP-2 Desktop % ./temp                                                            ]
0 3 6 9 12 15 18 21 %
michaelconti@Michaels-MBP-2 Desktop %
```

Usage with sizeof Operator

```cpp
int main() {
    size_t sizeOfInt = sizeof(int);
    size_t sizeOfDouble = sizeof(double);

    std::cout << "Size of int: " << sizeOfInt << " bytes\n";
    std::cout << "Size of double: " << sizeOfDouble << " bytes\n";

    return 0;
}
```



```
[michaelconti@Michaels-MBP-2 Desktop % ./temp                                                        ]
Size of int: 4 bytes
Size of double: 8 bytes
michaelconti@Michaels-MBP-2 Desktop % ▌
```

Lets Try it

- Modify
  - ✓ Class file (Point2D.cpp)
  - ✓ Header/Interface file (Point2D.h)
  - ✓ Driver (main.cpp)
- To use templating for any datatype