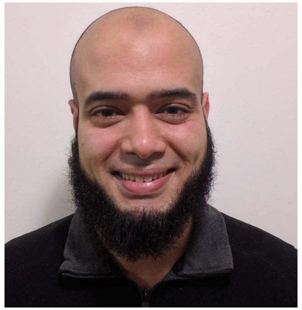
C++ Programming

Variadic Template

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



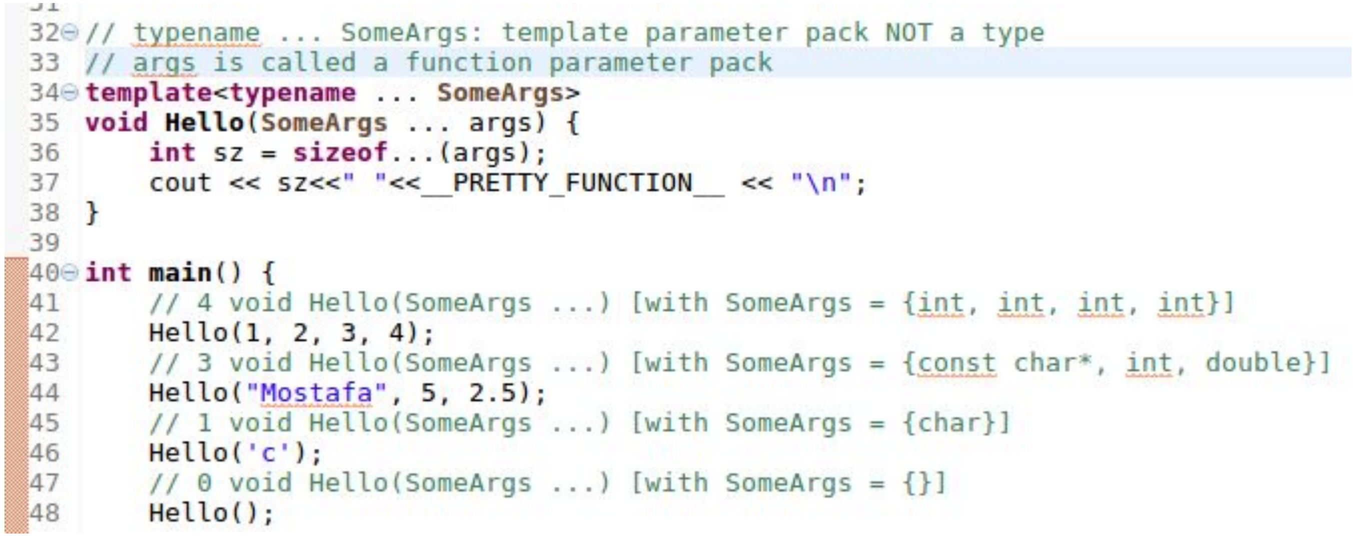
From fixed to variable # of arguments

- Our functions so far has a **fixed** #number of parameters
  - Int sum(int a, int b): 2 parameters of type int
    - You can also call sum(2, 3) but not sum(2, 5, 6, 7, 9) which **passed 5** arguments
    - Can we pass a variable number of arguments?
- Modern c++ allows a dynamic usage
  - C++11: Initializer list if they are of same type
  - C++11: Variadic Template
    - Typically provide 2 functions: One is recursive and another is base
  - C++17: Fold Expression
    - Simpler coding for specific 32 binary operators

Variadic

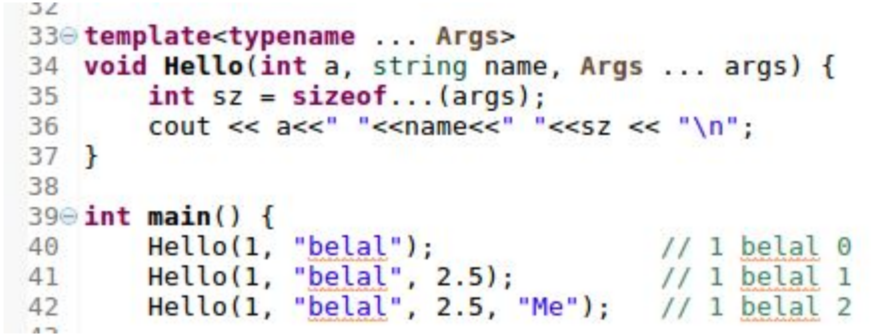
- The word Variadic is common in different languages
- **Variadic parameter** accepts **zero or more** values
    - Typically: a function may have **at most one** variadic parameter.
    - Typically: The right most parameter
    - Usually 3 periods (**ellipsis** syntax) are used to express it: **...**
- **Variadic functions** are functions which take a variable number of arguments
    - void printf(const char* fmt **...**) - OLD style - UNSAFE typically
- **Variadic template:** A modern & safe way to solve the problem

Parameter Pack



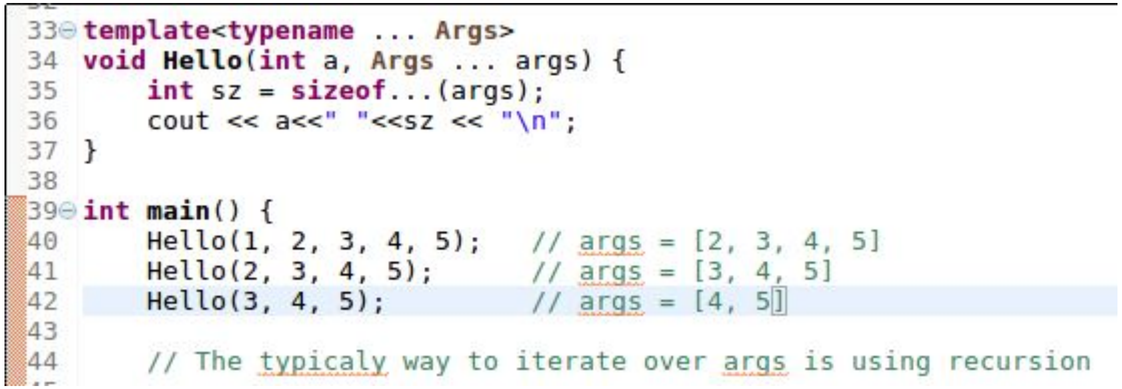
```cpp
// typename ... SomeArgs: template parameter pack NOT a type
// args is called a function parameter pack
template<typename ... SomeArgs>
void Hello(SomeArgs ... args) {
    int sz = sizeof...(args);
    cout << sz<<" "<<__PRETTY_FUNCTION__ << "\n";
}

int main() {
    // 4 void Hello(SomeArgs ...) [with SomeArgs = {int, int, int, int}]
    Hello(1, 2, 3, 4);
    // 3 void Hello(SomeArgs ...) [with SomeArgs = {const char*, int, double}]
    Hello("Mostafa", 5, 2.5);
    // 1 void Hello(SomeArgs ...) [with SomeArgs = {char}]
    Hello('c');
    // 0 void Hello(SomeArgs ...) [with SomeArgs = {}]
    Hello();
```

Parameter Pack



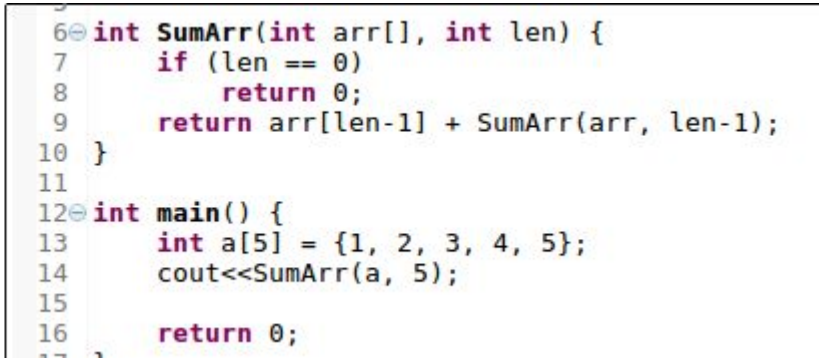
```
template<typename ... Args>
void Hello(int a, string name, Args ... args) {
    int sz = sizeof...(args);
    cout << a<<" "<<name<<" "<<sz << "\n";
}

int main() {
    Hello(1, "belal");                  // 1 belal 0
    Hello(1, "belal", 2.5);             // 1 belal 1
    Hello(1, "belal", 2.5, "Me");       // 1 belal 2
```

Parameter Pack



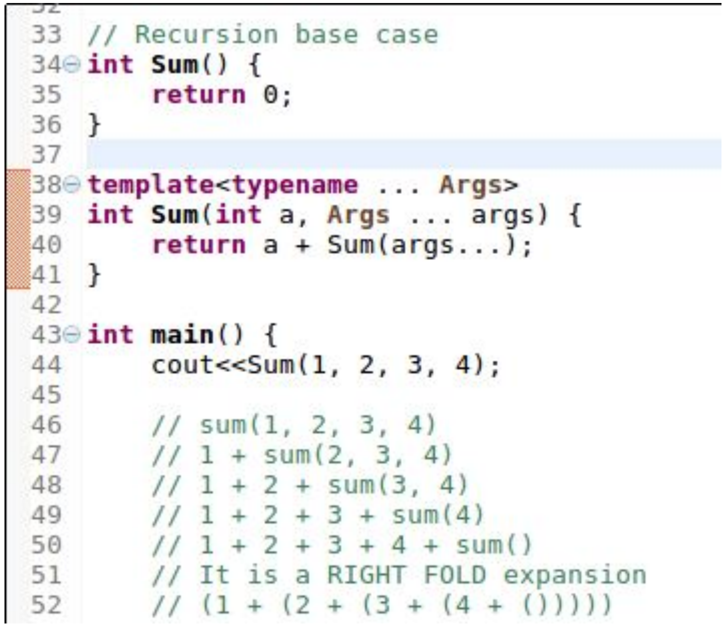
```
template<typename ... Args>
void Hello(int a, Args ... args) {
    int sz = sizeof...(args);
    cout << a<<" "<<sz << "\n";
}

int main() {
    Hello(1, 2, 3, 4, 5);   // args = [2, 3, 4, 5]
    Hello(2, 3, 4, 5);      // args = [3, 4, 5]
    Hello(3, 4, 5);         // args = [4, 5]

    // The typicaly way to iterate over args is using recursion
```

Recall: Array sum recursively



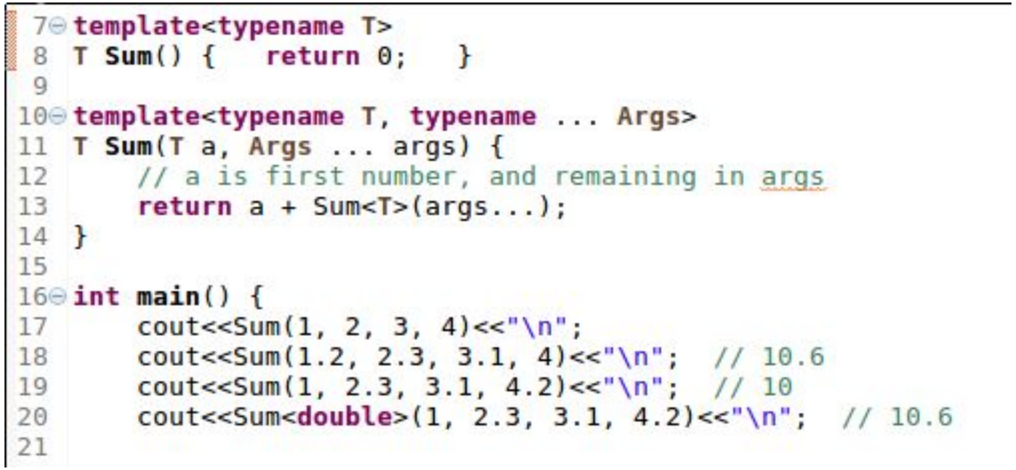
```
int SumArr(int arr[], int len) {
    if (len == 0)
        return 0;
    return arr[len-1] + SumArr(arr, len-1);
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    cout<<SumArr(a, 5);

    return 0;
```

Variadic Template



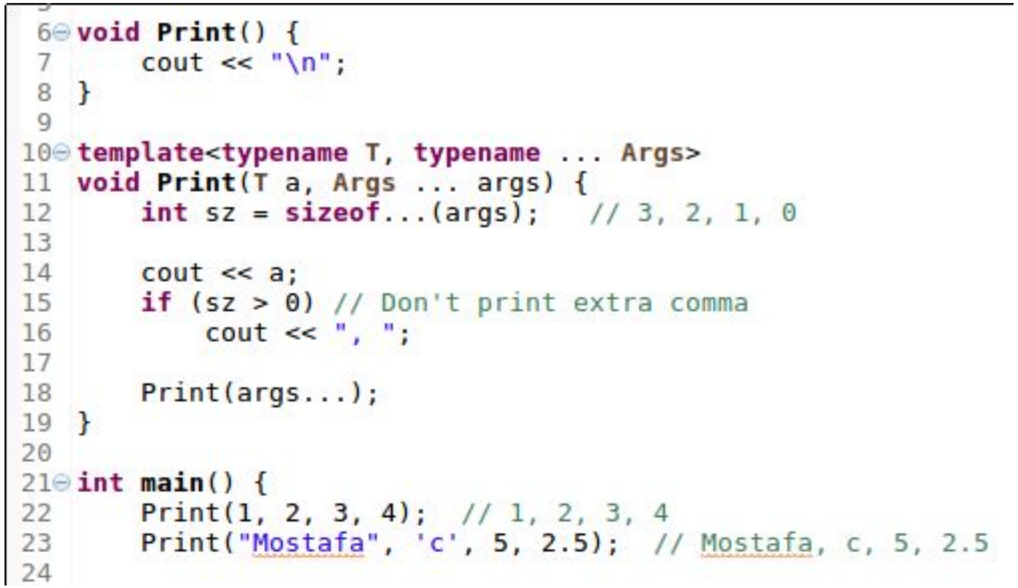
```
// Recursion base case
int Sum() {
     return 0;
}

template<typename ... Args>
int Sum(int a, Args ... args) {
     return a + Sum(args...);
}

int main() {
     cout<<Sum(1, 2, 3, 4);

     // sum(1, 2, 3, 4)
     // 1 + sum(2, 3, 4)
     // 1 + 2 + sum(3, 4)
     // 1 + 2 + 3 + sum(4)
     // 1 + 2 + 3 + 4 + sum()
     // It is a RIGHT FOLD expansion
     // (1 + (2 + (3 + (4 + ()))))
```

Variadic Template



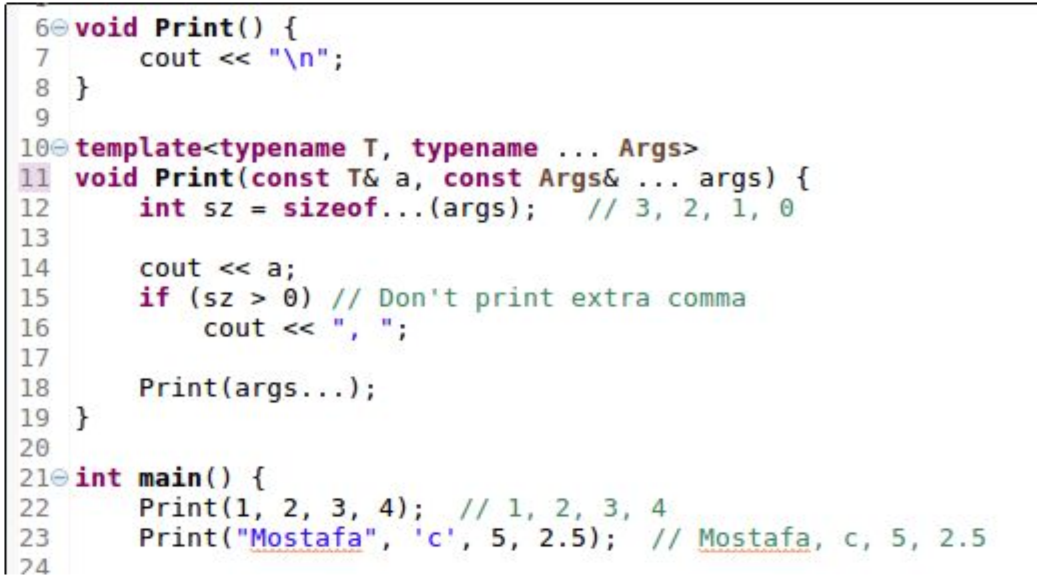
```cpp
template<typename T>
T Sum() {   return 0;   }

template<typename T, typename ... Args>
T Sum(T a, Args ... args) {
    // a is first number, and remaining in args
    return a + Sum<T>(args...);
}

int main() {
    cout<<Sum(1, 2, 3, 4)<<"\n";
    cout<<Sum(1.2, 2.3, 3.1, 4)<<"\n";  // 10.6
    cout<<Sum(1, 2.3, 3.1, 4.2)<<"\n";  // 10
    cout<<Sum<double>(1, 2.3, 3.1, 4.2)<<"\n";  // 10.6

```

Printing different types



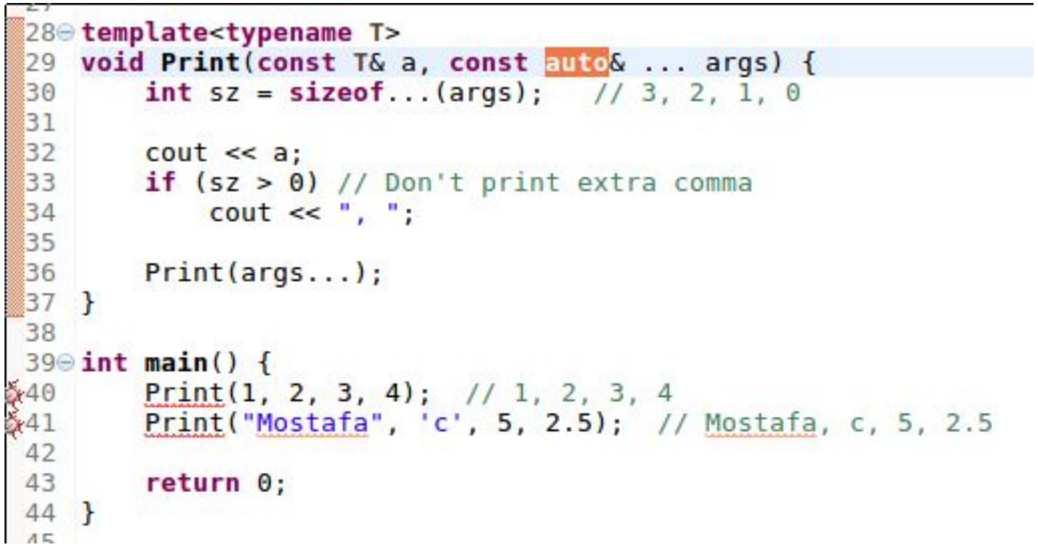
```cpp
void Print() {
    cout << "\n";
}

template<typename T, typename ... Args>
void Print(T a, Args ... args) {
    int sz = sizeof...(args);   // 3, 2, 1, 0

    cout << a;
    if (sz > 0) // Don't print extra comma
        cout << ", ";

    Print(args...);
}

int main() {
    Print(1, 2, 3, 4);  // 1, 2, 3, 4
    Print("Mostafa", 'c', 5, 2.5);  // Mostafa, c, 5, 2.5

```

Make const params



```
void Print() {
    cout << "\n";
}

template<typename T, typename ... Args>
void Print(const T& a, const Args& ... args) {
    int sz = sizeof...(args);   // 3, 2, 1, 0

    cout << a;
    if (sz > 0) // Don't print extra comma
        cout << ", ";

    Print(args...);
}

int main() {
    Print(1, 2, 3, 4);  // 1, 2, 3, 4
    Print("Mostafa", 'c', 5, 2.5);  // Mostafa, c, 5, 2.5
```

C++14: auto for the template parameter pack



```cpp
template<typename T>
void Print(const T& a, const auto& ... args) {
    int sz = sizeof...(args);   // 3, 2, 1, 0

    cout << a;
    if (sz > 0) // Don't print extra comma
        cout << ", ";

    Print(args...);
}

int main() {
    Print(1, 2, 3, 4);  // 1, 2, 3, 4
    Print("Mostafa", 'c', 5, 2.5);  // Mostafa, c, 5, 2.5

    return 0;
}
```

Generation

- Like any template-something, it is instantiated in **compilation** time!
- So compiler check out the use cases in your code
- Generate one code for each usage
- Ugly Compilation Errors

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."