# Variadic functions in C and C++

- We’ve seen variadic functions in lisp using &rest
- What about C/C++?
- C makes use of a set of macros (from stdarg.h)
- C++ makes use of templated functions
- What might the underlying implementation look like (i.e. on the system stack)?

# Internal implementation

- We usually find parameters at the top of the stack frame for a function call
- Suppose compiler simply adds one extra parameter at the very top that gives the number of parameters for the variadic part
- The callee then knows how many params it was passed
- As with most params, where the function has a positional reference to a variadic parameter, the compiler computes the correct offset to that parameter’s position and uses that

# Variadic functions in C

- Makes use of variety of preprocessor macros in stdarg.h
- Syntax for declaring function uses an int followed by ...

```
double f(int n, ...) // example f returning a double
```

- When calling the function we pass a count of how many other parameters are being passed, then the rest, e.g.

```
x = f(3, 27, 17, 8); // 3 is count of remaining params
```

- Most of the work is handled in the function implementation

# Function implementation

- Data type for argument list is `va_list`
- Functions to access the list are `va_start`, to initialize, and `va_arg`, to access next argument in list
- `va_end` is used to clean up at the end

```
double f(int n, ...) {
   va_list arglist;
   va_start(arglist, n); // sets up list storage
   // usage will go here
   va_end(arglist);  // deallocates argument list storage
}
```

# Accessing individual parameters

- access elements one at a time, in sequence, using va_arg
- Must pass argument list and expected data type
- Limited data types supported, ideally use long or double and type cast further if needed

```
double e; // expecting params to be doubles
e = va_arg(arglist, double); // get first, do whatever with it
for (int i = 1; i < n; i++) {
    e = va_arg(arglist, double); // get next
    // now can do whatever with e
}
```

# The C preprocessor solution

- the macros in stdarg.h actually rewrite the function calls prior to compilation
- functional, but clearly not an ideal solution, especially due to all the type casting

# Variadic functions in C++

- C++ uses techniques involving templated functions instead of the C-style use of stdarg.h
- requires two templated versions of the variadic function:
  - one base case with a fixed number of parameters (the minimum required number of parameters)
  - one with a variable number of parameters that processes a fixed number of the parameters and makes a recursive call
- The preprocessor uses the templates to build the necessary set of "real" functions that are run during execution

# Example: sum

- Function to take the sum of an arbitrary number of parameters, e.g.

```
x = sum(10, 1.5, 17.3, 300, 174);
x = sum(0.1, 23);
```

- Will write one templated version of sum that takes one parameter
- Will write one (recursive) templated version of sum that takes 1 parameter plus a variable number of others

# Basic (non-recursive) version

- Takes a single parameter, computes and returns result

```
template <typename T>
T sum(T x) {
   return x;
}
```

- Note that it’s templated, so can be any data type

# Recursive (general) case

- Take one fixed parameter, and number of additional args
- Express solution using a recursive call, need to template the type for the fixed parameter and the variadic type

```
template <typename T, typename... Args>
T sum(T x, Args... args) {
   return x + sum(args...);
}
```

- Note the ... uses carefully (easy syntax errors to make)

# Pros/cons

- templated, so more flexible w.r.t. data types
- data types used must still be compatible with data types used in the function implementations
- need to be able to express the solution as a recursive call with a smaller number of parameters (i.e. need to be able to "peel off" and process a fixed number of parameters with each call)