

# ECE326

## PROGRAMMING LANGUAGES

### **Lecture 21 : Variadic Functions and Template**

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

# Variadic Function

- Function with variable number of parameters
  - E.g. printf, scanf
  - Used when function also needs to deal with different argument types
- Syntax
  - ellipsis as last parameter of function signature

```
int eprintf(const char * fmt, ...);
```

# cstdarg

- Provides macro functions to extract arguments
- Limitation: requires a “pivot” argument
  - i.e. must have at least one known argument

```
// not ok -- cannot extract arguments using cstdarg  
int not_ok(...);
```

```
// ok -- cstdarg can use y as pivot argument  
int good(int x, int y, ...);
```

# cstdarg

```
#include <cstdarg> // provides va_list/va_start/va_end
#include <stdio.h> // contains vfprintf

int eprintf(const char * fmt, ...)
{
    va_list args; // stores variable argument list
    va_start(args, fmt);

    fprintf(stderr, "ERROR: ");
    // like fprintf, but takes va_list object instead of ...
    int ret = vfprintf(stderr, fmt, args);
    va_end(args);
    return ret;
}

eprintf("hello %s : %d\n", "world", 0); // ERROR: hello world : 0
```

# cstdarg

- `va_start(va_list ap, T pivot)`
  - Initialize ap with the pivot argument (can be of any type)
- `va_arg(va_list ap, T)`
  - Retrieves next argument and cast it to type T
- `va_end(va_list ap)`
  - End using ap and clean up resource

# Example

- Finds largest number out of  $n$  integers

```
int find_max(int n, ...) {  
    int i, val, largest;  
    va_list vl;  
    va_start(vl, n);  
    largest = va_arg(vl, int);  
    for (i = 1; i < n; i++) {  
        val = va_arg(vl, int);  
        largest = (largest > val) ? largest : val;  
    }  
    va_end(vl);  
    return largest;  
}
```

va\_arg is type-unsafe! It assumes the caller is passing in the expected type.

```
find_max(7, 702, 422, 631, 834, 892, 104, 772);    // 892
```

# Variadic Function in C

- Runtime solution
  - Variable argument processing occurs at runtime
- Type-unsafe
  - Compiler does not check if correct type is passed in
- Compile extension
  - `__attribute__((format(printf, 1, 2)))`
  - Allows for compile-time type-checking of printf arguments

# Variadic Template

- Template with variable number of parameters

```
template<typename T, typename... Args>
```

- Introduced in C++11
- Provides more type-safety by checking argument types
  - If pattern matching fails, code will not compile
- Enables many powerful templates
  - Recursive function/structure definitions
  - Function arguments forwarding
  - Template arguments forwarding



# Variadic Template Function

```
template<typename T>
bool is_in(T & a, T b) {
    return a == b;
}
```

Parameter pack



```
template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    return a == b || is_in(a, args...);
}
```

Pack expansion



```
/* read next character from file, see if it's a valid card */
char c = getc(file);
if (is_in(c, 'A', 'T', 'J', 'Q', 'K')) {
    return c;
}
```

# Variadic Template Function

```
template<typename T>           // base template
bool is_in(T & a, T b) {
    return a == b;
}
```

```
template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    return a == b || is_in(a, args...);
}
```

```
c = 'X'
is_in(c, 'A', 'T', 'J', 'Q', 'K')
```

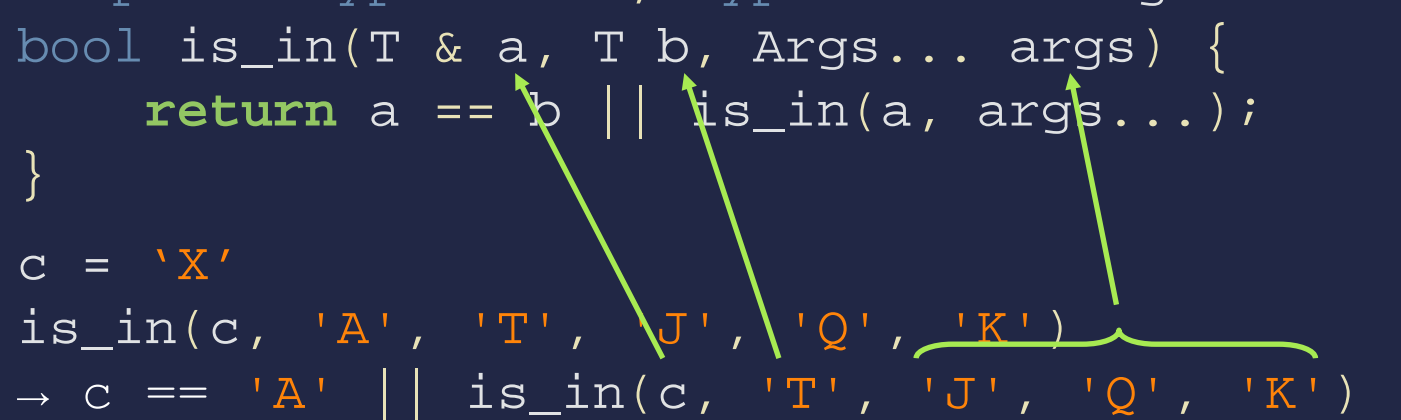
Pack expansion

# Variadic Template Function

```
template<typename T>           // base template
bool is_in(T & a, T b) {
    return a == b;
}

template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    return a == b || is_in(a, args...);
}

c = 'X'
is_in(c, 'A', 'T', 'J', 'Q', 'K')
→ c == 'A' || is_in(c, 'T', 'J', 'Q', 'K')
```

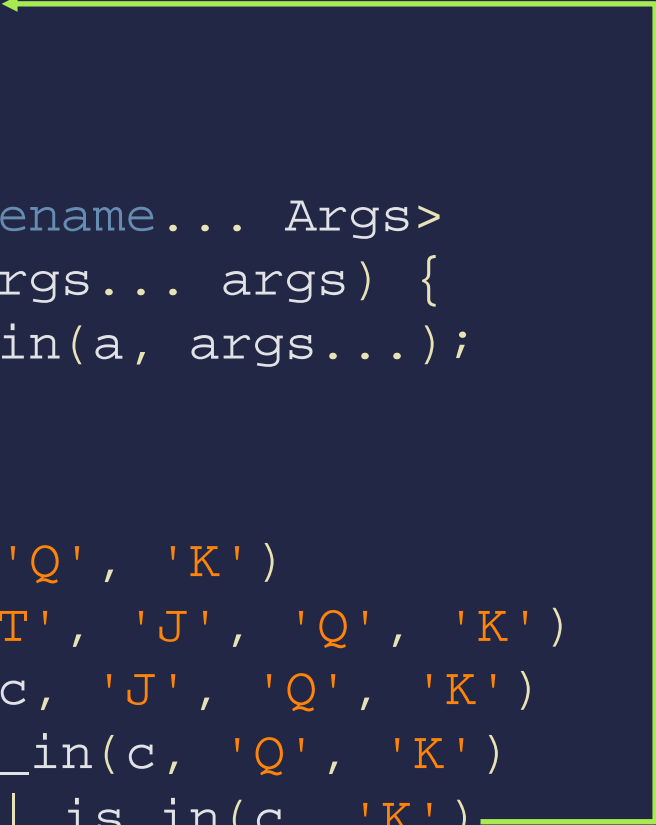


# Variadic Template Function

```
template<typename T>                // base template
bool is_in(T & a, T b) {
    return a == b;
}

template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    return a == b || is_in(a, args...);
}

c = 'X'
is_in(c, 'A', 'T', 'J', 'Q', 'K')
→ c == 'A' || is_in(c, 'T', 'J', 'Q', 'K')
    → c == 'T' || is_in(c, 'J', 'Q', 'K')
        → c == 'J' || is_in(c, 'Q', 'K')
            → c == 'Q' || is_in(c, 'K')
                → c == 'K'
```



# sizeof...

- Counts of the number of elements in parameter pack

```
template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    std::cout << "args contains " << sizeof...(args)
               << " elements\n";
    return a == b || is_in(a, args...);
}
```

args contains 4 elements  
args contains 3 elements  
args contains 2 elements  
args contains 1 elements  
X is not found

# Deduction Failure

- What if base case template is missing?

```
template<typename T, typename... Args>
bool is_in(T & a, T b, Args... args) {
    return a == b || is_in(a, args...);
}
```

```
is_in(c, 'A', 'T', 'J', 'Q', 'K')
→ c == 'A' || is_in(c, 'T', 'J', 'Q', 'K')
    → c == 'T' || is_in(c, 'J', 'Q', 'K')
        → c == 'J' || is_in(c, 'Q', 'K')
            → c == 'Q' || is_in(c, 'K')
                → c == 'K' || is_in(c)
```

error: no matching function for call to 'is\_in(char&)': return a == b || is\_in(a, args...);  
template argument deduction/substitution failed:  
candidate expects at least 2 arguments, 1 provided

# Function Argument Forwarding

- Allows forwarding arguments to another function
- Used extensively by std containers
  - E.g. `emplace_back` for `std::vector`
    - Calls constructor of the element type directly to avoid copying

```
// Example: wrapper function for diagnostics
template<typename RT, typename ... Args>
RT trace(RT (*func)(Args...), Args ... args) {
    cout << "Arguments: "; print_arguments(args ...);
    auto ret = func(args ...);
    cout << "Return value: " << ret << endl;
    return ret;
}
```

# Template Argument Forwarding

- Print the content of template containers
  - E.g. `std::vector`, `std::list`
  - These containers usually have an optional second parameter
    - Custom allocators are used for performance reasons

```
/* 1st parameter is a templated class with two parameters */
template <template <typename, typename> class ContainerType,
        typename T, typename Alloc>
void print_container(const ContainerType<T, Alloc>& c) {
    for (const auto& v : c) {
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```



# Template Argument Forwarding

- Works only if the template takes two parameters

```
vector<double> vd{3.14, 8.1, 3.2, 1.0};  
print_container(vd);  
list<int> li{1, 2, 3, 5};  
print_container(li);
```

- Problem

- Does not work for any other number of parameters
- E.g. unordered\_map (i.e. dictionary)
  - Takes 5 template parameters

```
map<string, int> msi{{"foo", 42}, {"bar", 81}, {"baz", 4}};  
print_container(msi);
```

# Catch-All Template

- Print the content of template containers
- Will take any number of template parameters
- Works as long as the container supports foreach loop

```
template <template <typename, typename...> class ContainerType,  
        typename T, typename... Args>  
void print_container(const ContainerType<T, Args...>& c) {  
    for (const auto & v : c) {  
        std::cout << v << ' ';  
    }  
    std::cout << '\n';  
}
```

# Recursive Template Class

- Inherits from another instantiation of itself

```
template<typename ... Names>
class JsObject {
protected:
    std::ostream & os;
    JsObject(std::ostream & os) : os(os) {}
};

template<typename T, typename ... Names>
class JsObject<T, Names...> : public JsObject<Names...> {
    std::string name;
public:
    JsObject(std::ostream & os, const T & name, Names ... names)
        : JsObject<Names...>(os, names...), name(name) {}
};
```

# Recursive Template Class

```
using S = const char *;  
auto jsobj = JsObject<S, S, S>(std::cout, "name", "age", "height");
```

	JsObject<>	JsObject<S>	JsObject<S, S>	JsObject<S, S, S>
jsobj	os	name= "height"	name= "age"	name= "name"