ECE326

PROGRAMMING LANGUAGES

Lecture 22 : 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
- Supported all the way back in C
- Denoted by the ellipsis syntax

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

- Custom-built variadic functions are type-unsafe
  - Type checking not done at compile time
    - Note: GCC extension `__attribute__((format(printf, 1, 2)))`

cstdarg

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

```
#include <cstdarg>    // provides variable argument handling
#include <cstdio>

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

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

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
- va_copy(va_list dst, va_list src)
  - Copy src to dst in its current state
    - May be halfway through the arguments when copied

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;
}

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

va_arg is type-unsafe! It assumes the caller is passing in the expected type.

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

Example

- From assignment 1 starter code, shoe.cpp

```
template<typename T>
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...);
}

/* 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;
}
```

Example

```
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...);
}

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'
```

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

emplace_back

- New method for std::vector in C++11
- Builds object directly within the vector
- Requires neither move or copy
  - In contrast, vector::push_back requires premade objects
- Requires forwarding arguments to constructor
  - Without a priori knowledge of constructor signature of type T

std::forward

- Similar to std::move, but for variable arguments
  - Syntax requires ... after the variable argument expansion

```
template<typename T, typename... Args>
T make_and_print(Args&& ... args) {
    /* create object of type T using forwarded arguments */
    T obj(std::forward<Args>(args)...);
    cout << obj << endl;
    return obj;
}

auto c = make_and_print<Complex>(5, 7);

5 + 7i
```

Example

- 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';
}
```

Example

- Works if the template only 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 4 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) {
        /* unordered_map returns std::pair during foreach loop,
         * therefore also need to implement “<<” for pair<T, U> */
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```