AP Capital *2020 Apr 20*

What are the differences between plain `enum` and `enum class`?

- There are three problems with plain `enum` :
- plain `enum` can be converted to integer implicitly, while `enum class` does not
- plain `enum` enumerators share the same scope as `enum` itself, while those of `enum class` are local to `enum class`
- plain `enum` underlying type cannot be specified, while that of `enum class` can be unsigned char, short or long

```
struct A
{
      enum status { s0, s1, s2 };
      enum class choice : unsigned short { c0, c1, c2 };
};

std::cout << A::s0 << A::status::s0;  // print 00 (as enum is converted to int, int is printed)
std::cout << A::s1 << A::status::s1;  // print 11
std::cout << A::s2 << A::status::s2;  // print 22
std::cout << A::c0;                   // compile error (as enumerator scope is inside enum)
std::cout << A::choice::c0;           // compile error (as enum class cannot be printed)
std::cout << int(A::choice::c0);      // print 0
std::cout << int(A::choice::c1);      // print 1
std::cout << int(A::choice::c2);      // print 2
```

Will `std:vector<T>` free memory after calling `std:vector<T>::clear()`?

- Out of my expectation ... lets try the following test

```
struct A
{
      A()         { std::cout << "\nA constructor"; }
      A(const A&) { std::cout << "\nA copy constructor"; }
      ~A()        { std::cout << "\nA destructor"; }
};

{
      A a0, a1, a2;
      std::vector<A> vec;
      std::cout << "\n============" << std::flush;
      vec.push_back(a0);  std::cout << " capacity = " << vec.capacity() << std::flush;
      vec.push_back(a1);  std::cout << " capacity = " << vec.capacity() << std::flush;
      vec.push_back(a2);  std::cout << " capacity = " << vec.capacity() << std::flush;
      std::cout << "\n------------" << std::flush;
      vec.clear();        std::cout << " capacity = " << vec.capacity() << std::flush;
      std::cout << "\n============" << std::flush;
}
```

Here is the printout, showing that destructor is called, while deletion is not. Total number of constructor called should be the same as that of destructor. Besides when vector is resized, multiple constructor / destructor calls are invoked.

```
A constructor                           // declaration of a0
A constructor                           // declaration of a1
A constructor                           // declaration of a2
============
A copy constructor       capacity = 1   // copy from obj a0 to vector[0]
A copy constructor                      // copy from old vector[0] to new vector[0]
A copy constructor                      // copy from obj a1 to new vector[1]
A destructor             capacity = 2
A copy constructor                      // copy from old vector[0] to new vector[0]
A copy constructor                      // copy from old vector[1] to new vector[1]
A copy constructor                      // copy from obj a2 to new vector[2]
A destructor
A destructor             capacity = 3
------------
A destructor
A destructor
A destructor             capacity = 3   // destructor called without deallocation, while reserved size remains 3
============
A destructor                            // destruction of a2
A destructor                            // destruction of a1
A destructor                            // destruction of a0
```

In the above red comment, we see that destructor can be called without deallocation. Conversely, allocation can be called before constructor, in other words allocation and construction can be decoupled. Consider declaring `std::vector<T>` object, memory is preallocated first without invoking any constructor not even default constructor, copy construction is delayed until `std::vector<T>::push_back(const T&)` is invoked.

What is placement new syntax?

- normally `new` operator does 2 things : heap-allocation and object-construction
- in fact, we can decouple the two steps, heap-allocation can be done prior to object-construction using new syntax
- placement new syntax is denoted as `A* ptr = new (allocated_buf) A[1000];`
- placement new syntax is useful for implementing custom-made container

```
struct A
{
      A(const X& x, const Y& y) {} // non-default constructor
};

// [method 1] normal new operator
A* ptr0 = new A[4]{{1,2}, {3,4}, {5,6}, {7,8}};                    // allocation and construction simultaneously

// [method 2] placement new syntax
char allocated_buf[sizeof(A)*4];                                   // allocation first
A* ptr1 = new (allocated_buf) A[4]{{1,2}, {3,4}, {5,6}, {7,8}};    // construction follows
```

What is allocator?

- Allocator is a template class that allows customized memory allocation for value type of containers.

```
template<typename T> struct allocator
{
      typedef T  value_type;
      typedef T* pointer;
      typedef T& reference;

      pointer allocate(int num_of_elements);
      void  deallocate(pointer ptr, int num_of_elements);
};
```

Can we adjust stack memory size?

- Stack size in linux can be checked by command `ulimit -s` displayed in kbytes.
- Stack size in linux can be changed by editing file `/etc/security/limits.conf`

When a process exits abnormally, will linux recycle all its resources? This is equivalent to asking whether leaked memory will be reclaimed or reused by other processes.

What are the differences between `std::string::data()` and `std::string::c_str()`?

- Both returns `char*` pointing to array of characters
- `data()` may or may not contain a null character
- `c_str()` must contain a null character

Implement depth first search in a binary tree (not binary search tree)?

```
template<typename T> struct node
{
	T value;
	node<T>* lhs = nullptr;
	node<T>* rhs = nullptr;
};

template<typename T> node<T>* dfs(node<T>* this_node, const T& target)
{
	if (this_node == nullptr) return nullptr;
	std::stack<node<T>*> s; s.push(this_node);
	while(!s.empty())
	{
		if (s.top()->value == target) return s.top();
		s.pop();
		if (this_node->lhs != nullptr) s.push(this_node->lhs);
		if (this_node->rhs != nullptr) s.push(this_node->rhs);
	}
	return nullptr;
}
```

What are the problems with the following multithreading code?

```
struct foo { int x = 0; int y = 0; };

foo f;
std::thread t0([&](){ for(int n=0; n!=10000; ++n) ++f.x; })
std::thread t1([&](){ for(int n=0; n!=10000; ++n) ++f.y; })
t0.join();
t1.join();
```

- There is no racing condition as threads are accessing different resources.
- There is performance issue with false sharing, as integers are mapped to same cache line, leads to ping pong.

What are the differeces between TCP socket in boost and in BSD (i.e. raw socket)? What is IO multiplexing?