ECE326

PROGRAMMING LANGUAGES

Lecture 13 : Generic Programming

Kuei (Jack) Sun
ECE
University of Toronto
Fall 2019

Container

- A collection of objects
- Commonly used containers
  - Dynamic arrays
    - Constant time random access, best performance when used as stack
  - Doubly Linked List
    - Constant time deletion, best performance when used as queue
  - Map
    - Stores key-value pairs
  - Priority Queue
    - Automatically sorts data during insertion

Generic Container

- In C, typically done using `void *` pointers

```
struct vec1 {
    void ** array;
    int bytes;
    int count;
};
```

Sample usage:

```
struct vec1 * v  = new_vector1();
struct point * p = new_point(1, 2);
v1append(v, p);
…
p = (struct point *)v1get(v, 1);
```

```
int v1append(struct vec1 * v, void * element) {
    if (v->count >= v->bytes/sizeof(void *)) {
        if (!(v->array = realloc(v->array, v->bytes *= 2)))
            return –ENOMEM;
    }
    v->array[v->count++] = element;
    return 0;
}
```

Generic Container

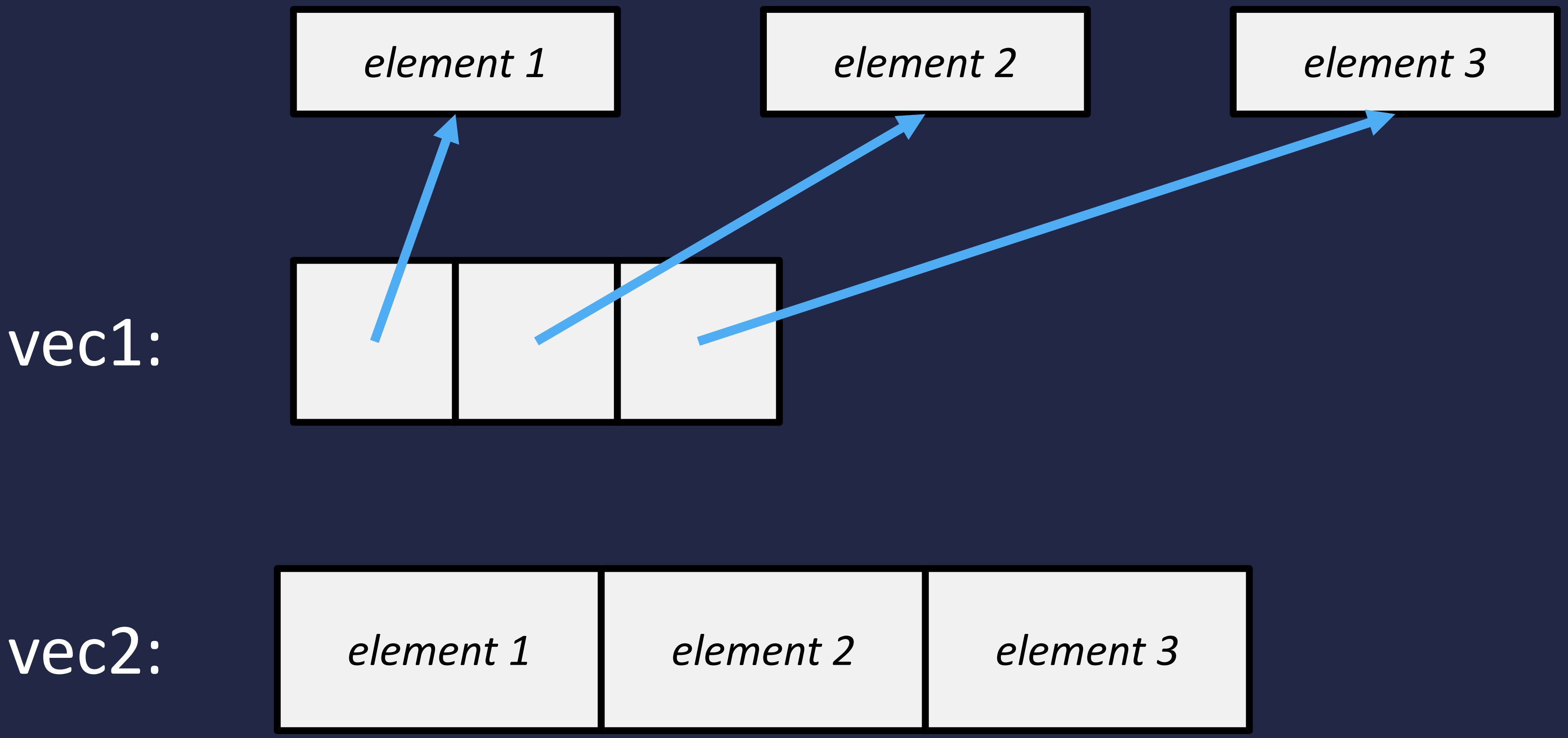
- Specially managed buffer (element embedded in buffer)

```
struct vec2 {
    char * buffer;
    int bytes;
    int count;
    int size;
};
```

```
void * v2get(struct vec2 * v, int i) {
    if (i >= count) return NULL;
    return (void *)(v->array + i*v->size);
}
```

```
int v2append(struct vec2 * v, void * element) {
    if ((v->count+1)*v->size > v->bytes) {
        if (!(v->buffer = realloc(v->buffer, v->bytes *= 2)))
            return –ENOMEM;
    }
    memcpy(v->buffer+(v->count++)*v->size, element, v->size);
    return 0;
}
```

Generic Container



Linux Doubly Linked List

- Used extensively by Linux Kernel

```
struct list_head {
    struct list_head *next, *prev;
};

struct foo {
    int x, y;
    struct list_head node;
};

struct bar {
    int a, b;
    struct list_head head;
};
```

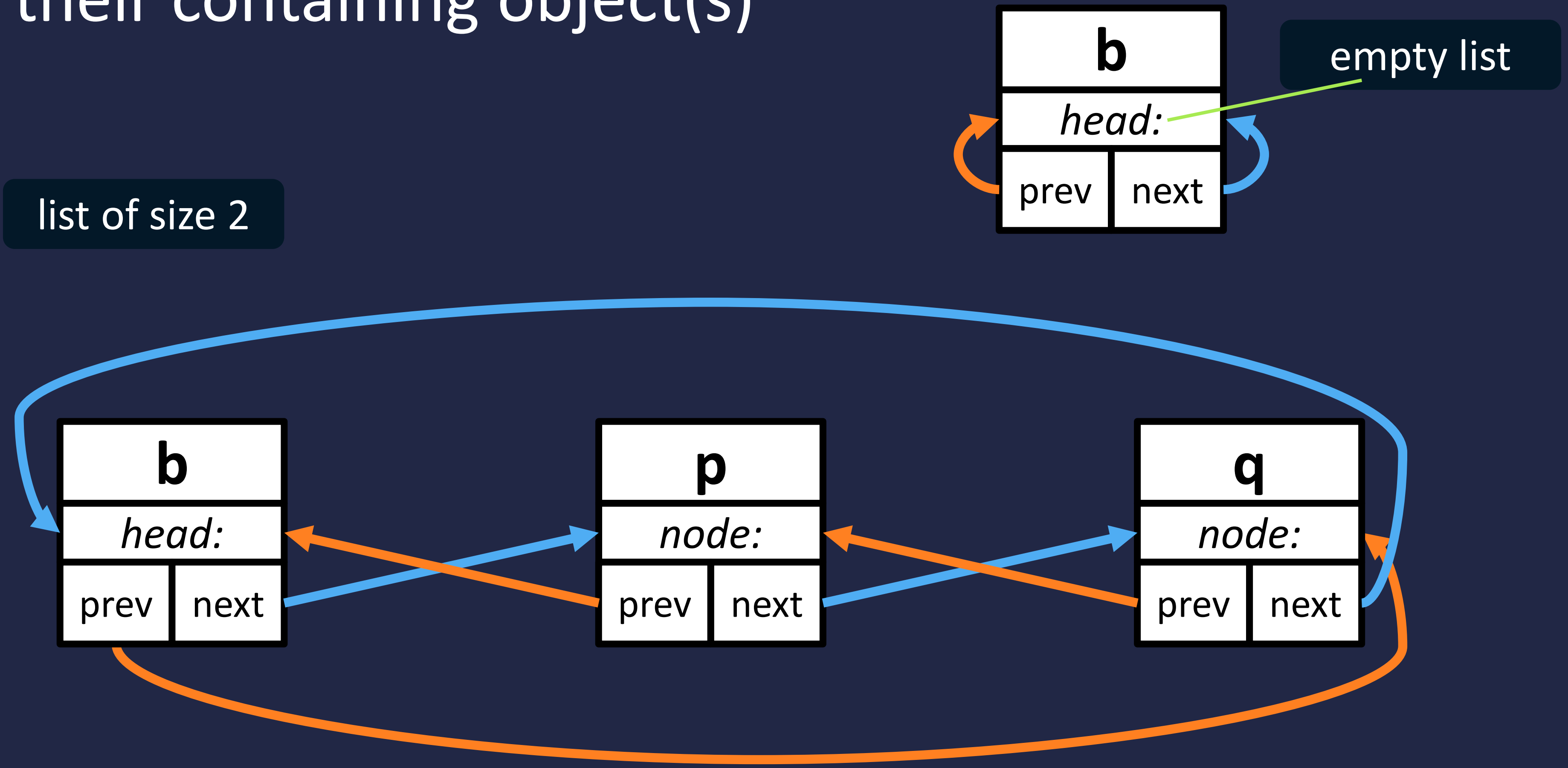
```
struct bar b = { 0 };
INIT_LIST_HEAD(&b.head);
struct foo q, p;
list_add(&q.node, &b.head);
…
// get the item at head of list
struct foo * f;
f = list_first_entry(
    &b.head, struct foo, node
);
```

Circular Linked List

```
typedef struct list_head {
    struct list_head *next, *prev;
} LH;

static inline void INIT_LIST_HEAD(LH *list) {
    list->next = list;
    list->prev = list;
}

#define list_add(new, head) \
    __list_add((new), (head), (head)->next)

static inline void __list_add(LH *new, LH *prev, LH *next) {
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

Circular Linked List

- list_head structures point to themselves, and not to their containing object(s)



container_of

- How does list_first_entry cast list_head to struct foo?

```
struct foo * f = list_first_entry(&b.head, struct foo, node);

#define list_first_entry(ptr, type, member) \
      list_entry((ptr)->next, type, member)

#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *)( (char *)__mptr - offsetof(type, member) ); \
})
```

typeof

- Returns type of expression (at compile time)

```
int x = 5;
typeof(&x) y;     // y is of type “int *”

#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *)( (char *)__mptr - offsetof(type, member) ); \
})
```

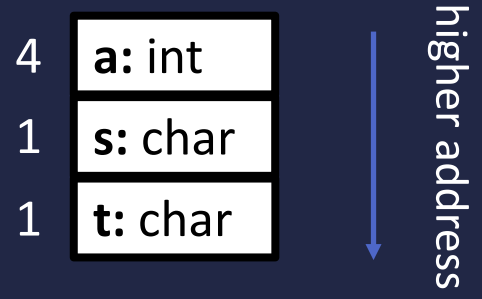
- `container_of((&b.head)->next, struct foo, node)` expands to:

```
typeof(((struct foo *)0)->node) *__mptr = ((&b.head)->next);
(struct foo *)((char *)__mptr - offsetof(struct foo, node));
```

offsetof

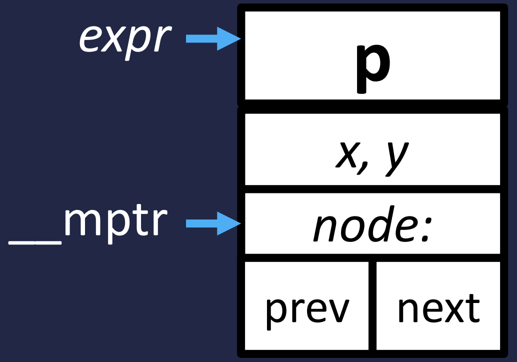
- Returns byte offset of member variable in structure

```
#define offsetof(TYPE, MEMBER) \
    ((size_t) &((TYPE *)0)->MEMBER)
struct baz { int a; char s; char t; };
// prints 5
printf(“%d”, offsetof(struct baz, t));
```




```
typeof(((struct foo *)0)->node) *__mptr = ((&b.head)->next);
(struct foo *)((char *)__mptr - offsetof(struct foo, node));
```

```
struct list_head * __mptr = ((&b.head)->next);
/* this is the value of the expression */
(struct foo *)((char *)__mptr – 12;
```



Generics in C

- Use generic pointer (void *)
- Use generic buffer
- Use container_of to reference arbitrary parent object

- Lots of low level pointer manipulation
- Type-unsafe
- Easy to make mistakes

Generic Programming

- Writing program that makes minimal assumption about the structure of data
  - Maximize code reuse across different data types
- Parametric polymorphism
  - Ability to handle values without depending on their types
    - Concatenating two lists can be done without knowing type of element
- Metaprogramming
  - Writing program that modifies programs
  - Generics often implemented with metaprogramming

Parametric Polymorphism

- Problem with statically typed languages
  - Redundant implementation of common algorithm

```
int min(int a, int b) {
    return a < b ? a : b;
}
```

```
/* assumed operator< implemented */
Complex min(Complex a, Complex b) {
    return a < b ? a : b;
}
```

- Imagine if common code is complex (like a container)
  - Tedious
  - Time consuming
  - Error prone (one mistake means fixing all other versions)

Generics in Java

- Before J2SE 5.0

```
List v = new ArrayList();
v.add(“hello world”);   // inserting a string into v
Integer i = v.get(0);   // ERROR: runtime type error
```

- After
  - Compile-time type checking added to generics
  - Underlying implementation remains same (*shared generics*)

```
List<Integer> v = new ArrayList<Integer>();
v.add(“hello world”);   // ERROR: cannot add String
v.add(5);               // OK: can add int to Integer list
Integer i = v.get(0);   // OK: list guaranteed to store Int
```

Generics in C++

- Template Metaprogramming
  - For each template instantiation, new code is generated

```
vector<int> v;
v.push_back(5);           // insert 5 into list
cout << v[0] << endl;     // print first element of list (5)

vector<float> f;          // another instance of template
f.push_back(2.3);
```

- Can seriously bloat size of program
  - Abuse of templates can result in very large executable
  - Need proper balance with Java/C's approach

C++ Template Programming

an introduction

Template Function

- Behaves like function except function is created when function is used (template instantiation)
- Template must fully implement (define) function
  - Cannot just declare the function (why?)

```
template<typename T>
T min(T a, T b) {
    return a < b ? a : b;
}

std::cout << min<int>(4, 9); // prints 4
std::cout << min(3, 7);  // can omit <int> (type inferred)
```

template parameter T
T is called *parameterized type*
(type that is a parameter)

C++ Template

- Type-safe at compile time
- Template instantiated (code generated) on use
- Type must support all used operations in template
  - E.g. operator< must be supported by type T

```
template<typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

  - If not, must use template specialization

Template Specialization

- Allows alternative implementation for a particular type
- Benefit
  - Code does not make it to executable if not used!

```
template<typename T>
T min(T a, T b) {
    return a < b ? a : b;
}

/* use strcmp to compare cstrings */
template<>
const char * min(const char * a, const char *b) {
    return strcmp(a, b) < 0;
}
```

Multiple Parameters

- Templates can have multiple parameters
  - Instantiation may require disambiguiation

```
/* template declaration – must define within same file */
template<typename T, typename F> T convert(F v);
template<typename A, typename B> A distance(A a, B b);

double foo(double d) {
    /* C++ does *not* infer from return type */
    int i = convert<int>(d);        // from double to int
    char c = convert<char>(i);      // from int to char
    /* instantiate convert<double, char> */
    double(*func)(char) = convert;  // function pointer
    return distance(func(c), 5);
}
```

Other Parameters

- C++ templates do not restrict on parameter type

```
template<typename T, int A, int B>
vector<T> & partial_sort(vector<T> & v) {
    if (v.size() < A) return v;
    int max = v.size();
    if (max >= B) max = B;
    for (int i = A; i <= max-1; i++) {
        for (int j = i+1; j <= max; j++) {
            if (v[j] < v[i])
                swap(v[j], v[i]);   // pass by reference
        }
    }
    return v;
}
```

Function Overloading

1. Non-template function overload
2. Template specialization
3. More specific and specialized template
4. Base template

```
template<class T> void f(T);        // #1: template overload
template<class T> void f(T*);       // #2: template overload
void                   f(double);   // #3: non-template overload
template<>        void f(int);      // #4: specialization of #1

f('a');          // calls #1, f<char>(char)
f(new int(1));   // calls #2, f<int *>(int *)
f(1.0);          // calls #3, f(double)
f(1);            // calls #4, f<>(int)
```