CS 211 : Thurs 02/01 (lecture 09)

Prof. Hummel
(he/him)

- **<u>Topics</u>: intro to C++**

Notes:

- *Lecture slides available on Canvas*
- ***HW 04*** *due tonight (Thursday)*
- ***Project 04*** *due Sunday night – let's add the late period back, you can submit as late as Tues night*
- ***HW 05*** *(intro to C++) next Thursday 2/8 --- no project due next week unless you want to work on extra credit project (release Monday, due Friday)*

Northwestern University

What's the error in this code fragment?

```
struct RAM_VALUE* value = (struct RAM_VALUE*) malloc(sizeof(struct RAM_VALUE));

assert(expr != NULL);
assert(expr->lhs != NULL);  // an expr always has a LHS
assert(expr->lhs->expr_type == UNARY_ELEMENT); // no unary operators (yet)

value = get_element_value(stmt, memory, expr->lhs->element);

if (value == NULL)  // semantic error?
 return false;
```

A) Did not malloc enough memory

B) An expr may not have a left-hand side (LHS)

C) get_element_value( ) should return a struct, not a pointer to a struct

D) Memory leak

Helper function mallocs… Which caller is correct?

```c
char* get_user_input(struct ELEMENT* parameter) {
  assert(parameter->element_type == ELEMENT_STR_LITERAL);

  printf("%s", parameter->element_value);

  char line[256];
  fgets(line, sizeof(line), stdin);
  line[strcspn(line, "\r\n")] = '\0';

  size_t bytes = sizeof(char) * (strlen(line) + 1);
  char*  input = (char*) malloc(bytes);

  strcpy(input, line);
  return input;
}
```

(A)

```c
char* input = get_user_input(call->parameter);

struct RAM_VALUE value;
value.value_type = RAM_TYPE_STR;
value.types.s    = input;

ram_write_cell_by_id(memory, value, var_name);
```

(B)

```c
char* input = get_user_input(call->parameter);

struct RAM_VALUE value;
value.value_type = RAM_TYPE_STR;
value.types.s    = input;

ram_write_cell_by_id(memory, value, var_name);

free(input);
```

(C)

```c
char* input = get_user_input(call->parameter);

struct RAM_VALUE* value;
value->value_type = RAM_TYPE_STR;
value->types.s    = input;

ram_write_cell_by_id(memory, *value, var_name);

free(value);
free(input);
```

C != C++

View C++ as a completely new language…

Why C++

- **Goals:**
  - *Object-oriented (OOP)*
  - *Higher-level abstractions*
  - *Dynamic data structures*
  - *Executes as fast as C using modern programming style*

You can write C code if necessary --- C++ is backward-compatible and supports C.

Most C++ programmers don't write C, and avoid pointers.

Demos

- Login to replit.com
- Open team…
- Open project "Lecture 09"

C

Programming

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int  x, y, z;

  printf("Enter 2 integers> ");
  scanf("%d", &x);
  scanf("%d", &y);

  z = x + y;

  printf("sum=%d\n", z);

  return 0;
}
```

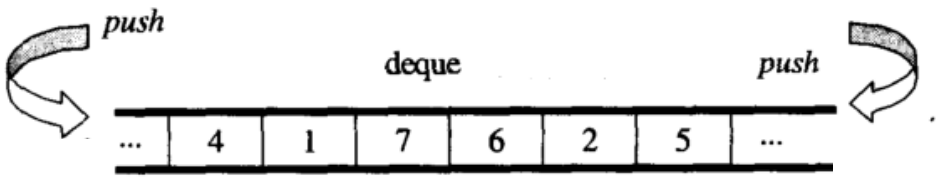
```cpp
#include <iostream>

using namespace std;

int main()
{
   int  x, y, z;

  cout << "Enter 2 integers> ";
  cin >> x;
  cin >> y;

  z = x + y;

  cout << "sum=" << z << endl;

  return 0;
}
```

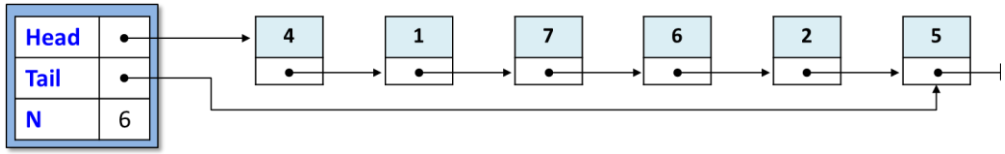
C++ abstractions (part 01)

- **objects**
- **templates**
- **foreach**
- **[ ]**

Recall C-based deque data structure

- **Deque ("deck")**
  - *Abstraction:*
  - *Implementation:*

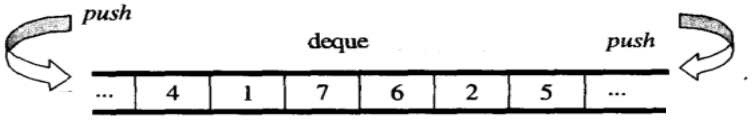





```
struct IntDeque* intdeque_create(void);

void intdeque_destroy(struct IntDeque* dq);

void intdeque_push_front(struct IntDeque* dq, int value);

void intdeque_push_back(struct IntDeque* dq, int value);

int  intdeque_size(struct IntDeque* dq);

bool  intdeque_get(struct IntDeque* dq, int position, int* value);
```

```
struct IntDeque {
  struct IntNode* head; // 1st element
  struct IntNode* tail; // last element
  int             N;    // # of values
};
```

```
struct IntNode {
  int             value;
  struct IntNode* next;
};
```




```c
#include "intdeque.h>

int main()
{
  struct IntDeque* dq = intdeque_create();

  intdeque_push_front(dq, 123);
  intdeque_push_back(dq, 456);
  intdeque_push_back(dq, 789);

  for (int i=0; i<intdeque_size(dq); i++) {
    int value;
    intdeque_get(dq, i, &value);
    printf("%d\n", value);
  }

  printf("\n");

  struct IntNode* cur = dq->head;
  while (cur != NULL){
    printf("%d\n", cur->value);
    cur = cur->next;
  }
```

```cpp
#include <deque>

using namespace std;

int main()
{
  deque<int>  dq;

  dq.push_front(123);
  dq.push_back(456);
  dq.push_back(789);

  for (int i : dq)  // foreach:
    cout << i << endl;

  cout << endl;

  for (size_t i=0; i<dq.size(); i++)
    cout << dq[i] << endl;
```

C++ abstractions (part 02)

- **Memory management**
  - *Constructors*
  - *Destructors*

C

Programming

```c
#include "intdeque.h>

int main()
{
  struct IntDeque* dq = intdeque_create();

  intdeque_push_front(dq, 123);
  intdeque_push_back(dq, 456);
  intdeque_push_back(dq, 789);

  .
  .
  .

  intdeque_destroy(dq);

  return 0;
}
```

In C, you have to call the create & destroy functions explicitly…

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main()
{
  deque<int>  dq;

  dq.push_front(123);
  dq.push_back(456);
  dq.push_back(789);

  .
  .
  .

  return 0;
}
```

Special "constructor" function is implicitly called to create object

Special "destructor" function is called to destroy object

C++ abstractions (part 03)

- **strings**

C

Programming

```c
#include <string.h>

int main()
{
  char* s1 = "apple, ";
  char* s2 = "banana";

  size_t bytes = sizeof(char) *
    (strlen(s1) + strlen(s2) + 1);

  char* result = (char*) malloc(bytes);

  strcpy(result, s1);
  strcat(result, s2);
  printf("%s\n", result);

  .
  .
  .

  free(result);
  return 0;
}
```

String concatenation…

```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s1 = "apple, ";
  string s2 = "banana";

  string result = s1 + s2;

  cout << result << endl;

  .
  .
  .

  return 0;
}
```

Destructors are called to free memory inside each string…

C++ abstractions (part 04)

- **pass-by-reference**

From Project 03…

- **One possible design for get_element_value( )…**

```
//
// get the element's value, which might be an identifier or a literal. Returns true
// if successful, returning the type and value via the parameters.
//
bool get_element_value(struct STMT* stmt, struct RAM* memory, struct ELEMENT* element,
                       int* value_type, int* i, double* d, char** s)
{
  if (element->element_type == ELEMENT_INT_LITERAL)
  {
     *value_type = RAM_TYPE_INT;
     *i = atoi(element->element_value);
     return true;
  }
  else if (…)
  .
  .
  .
}
```

```
int    value_type;
int    i;
double d;
char*  s;

bool success = get_element_value(stmt, memory, lhs->element, &value_type, &i, &d, &s);
```

Pass-by-reference

- **This form of parameter passing is called "pass-by-ref"**
  - *C++ supports this via the & operator,* ***creating ptrs for you***

```
//
// get the element's value, which might be an identifier or a literal. Returns true
// if successful, returning the type and value via the parameters.
//
bool get_element_value(struct STMT* stmt, struct RAM* memory, struct ELEMENT* element,
                       int& value_type, int& i, double& d, string& s)
{
  if (element->element_type == ELEMENT_INT_LITERAL)
  {
     value_type = RAM_TYPE_INT;
     i = atoi(element->element_value);
     return true;
  }
  else if (…)
  .
  .
  .
}
```

```
int     value_type;
int     i;
double  d;
string  s;

bool success = get_element_value(stmt, memory, lhs->element, value_type, i, d, s);
```

What should I be working on?

HW #04 *due tonight*

Project #04 *due Sunday night (can submit late, up until Tuesday night)*