Lecture 15: Invariants

An invariant is a set of conditions that must always evaluate to true at certain well-defined points; otherwise, the program is incorrect.

Representation Invariant

For ADT, it is called representation invariant.

It describes the conditions that must hold on those members for the representation to correctly implement the abstraction. These conditions are not naturally preserved, but need your caution in implementing the class.

Establishing the Invariant

Each method in the class can assume that the invariant is true on entry if:

- The representation invariant holds immediately before exiting each method
- Each data element is truly private.

Therefore, the class is self-consistent.

Essentially, for each method, you should:

- Do the work of the method (implement)
- Repair the invariants you broke (if any)

Checker Function

defensive programming: write a private method to check whether all invariants are true (before exiting, or after entering, each method):

```
bool repOK();
// EFFECTS: returns true if the rep. invariants hold
```

Next, add the following code right before returning from any function that modifies any of the representation: `assert(repOK())`;
you can write the same line at the beginning of every method too, to check whether the assumption the method relies on is true.

Exercise 1

How should `repOK()` be? What invariant should `load(..)` repair?

```
class buckets{
    unsigned int buckets[MAX_SIZE];//buckets initially empty to be loaded
    unsigned int firstLoadedBucket;//the index of first loaded bucket
    unsigned int bucketLoadedNum;//the number of loaded bucket
    unsigned int loadNum;//the amount of load
    double loadFactor;//(loadNum/MAX_SIZE)
    bool repOK();// EFFECTS: returns true if the rep. invariants hold
    ...
  public:
    void load(unsigned int bucket_index, unsigned int load);
    ...
```

Lecture 16: Dynamic Memory Allocation; Overloading,Default Arguments; Destructor

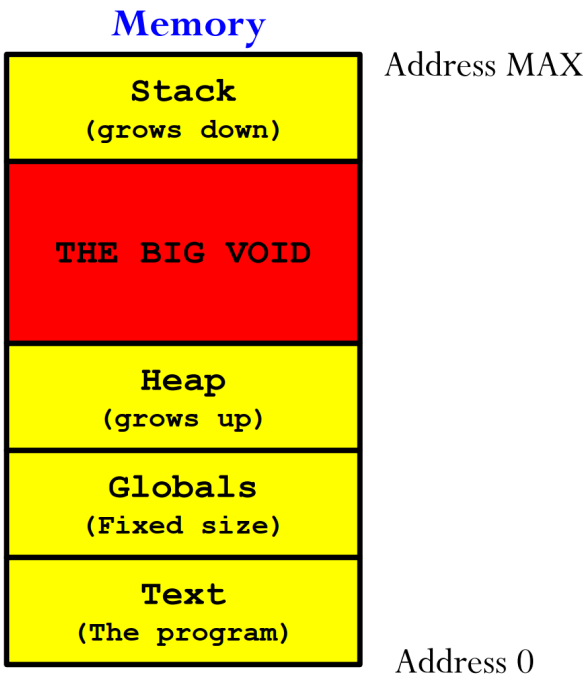
Dynamic Memory Allocation

Types of Variables:

There are basically three types of variables in C++ programs:

- Global Variables (static)
    - Defined outside functions
    - Lifetime: Their space is set aside before the programs runs (at compile time), and is reserved until the program terminates.
- Local Variables (static)
    - Defined within a block (basically insides curly brackets {})
    - Lifetime: Space is set aside when relevent block is entered (at run time), and reserved until the block is exited.
- Dynamic Variables
    - the compiler doesn't need to know how big it is and how long it lives.
    - Space is allocated using new, and deallocated using delete by programmer.

Memory Structure:



Text stores the code, heap stores dynamic variables and stack stores memory for functions (including main function and local variables).

When stack and heap use up the big void, the program overflows. How many memory they use is indicated by a pointer pointing to the top of heap and another pointing to the bottom of stack.

The "Stack" and "Heap" here refers to specific locations in the memory. They are not equivalent to the data structures called "stack" and "heap", though their behavior is similar.

Dynamic Memory

Why dynamic memory?

- Local variables can only be fixed size, we want to change the size at runtime.

```
int length;
cin >> length;
int arr[length];
```

This code may be compiled successfully with some compilers. However, it is not allowed in ISO c++ standard. Some compilers allow this as an extension. You can add -pedantic flag to turn on the warning.

- We don't know the exact lifetime of the objects.

Allocation

```
Type* obj0 = new Type; // Default construction
Type* obj1 = new Type(); // Default construction
Type* obj2 = new Type(arg1, arg2); // Constructor with 2 params
Type* objA0 = new Type[size]; // Default cons each elt
Type* objA1 = new Type[size](); // Same as obj A0

int* a=new int;//see 'int' as a class
int* b=new int(1);
int* c=new int[5];
int* d=new int[5]{0};//[0,0,0,0,0]
```

However, you cannot allocate an array of objects using non-default constructors.
When using `new` and `new[]`, following things happens:

1. Allocates space in heap (for one or an array of objects).
2. Constructs object in-place (mainly be calling constructor).
3. Returns the "first" address.

You may remeber `malloc` and `free` in the C language. They only deal with the memory while do nothing to the content in it. That means in C you need to malloc the space and initialize it by hand.
Deallocation
Use `delete` and `delete[]` to deallocate single object and arrays respectively.

Memory Leak
The usage of new/delete is very easy. The difficult point is when and where to use them. Basically, memory leak happens when you lose the address of some dynamic memory (then it would be impossible for you to delete that memory).

Example:

```
// Each time foo() is called, there is new memory allocated.
// And since p is a local variable, each time p will point to a new address
void foo() {int* p = new int(0); /* Code */}
```

Each time `foo()` is called, some memory is occupied and will not be released even after the program terminates. Gradually, you will drain out all memory of your computer, which is very bad.

Check Memory Leak

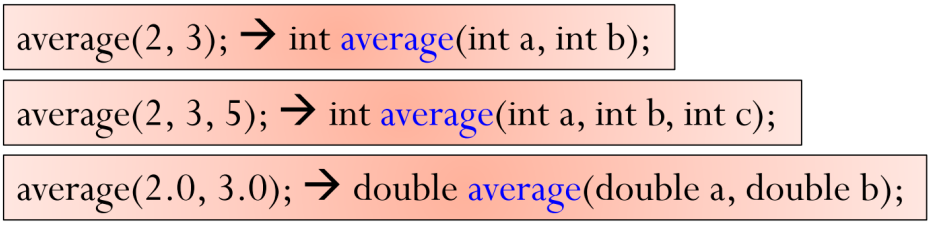
1. **valgrind**
   Command: `valgrind --leak-check=full ./program <args>`
   Function: search for memory leaks and give details of each individual leak.
   To install, type the command: `sudo apt-get install valgrind`
2. **fsanitize**

   compile the program using `g++ -o program <file> -fsanitize=address` and run.

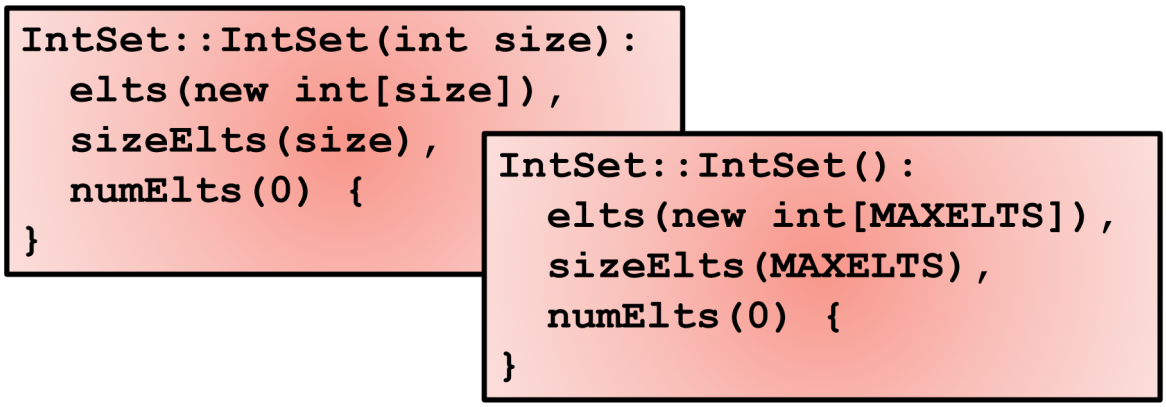
Overloaded Constructor and Default Argument

Overloaded Constructor

Functions with same name can have different versions, and compiler tells which function to call based on the actual argument count and types



For constructor, programmer can choose default initialization (eg. MAXELTS) or sepcialized (eg. give a size).



Default Argument

There's a easier method, using default argument.

Default argument means when you put in no argument, the function can assume a default argument

for example, `IntSet(int size = MAXELTS);` will realize the function of overloaded constructor with just one function.

Mind that you should put default argument to the end

```
int add(int a, int b = 0, int c = 1); // OK
int add(in a, int b = 1, int c); // Error
```

Also mind that you don't need to write default argument when implementing.

Destructor

Class's member is dynamic

When all the members of a class are static, there's no need to write the destructor, since there's a default one that help the compiler automatically released memory of the class's members when the program ends.

However when there are some dynamic members in a class, the dynamic members can't be automatically released, and that's where we need a destructor function to help us released them at once.

Class is dynamic

The destructor can be called automatically when a class's life ends (like main function or other function ends). But if you dynamically declare a class, it of course should be deleted by programmer, since a dynamic variable can't be deleted by compiler and its destructor won't be called.

Exercise 2

How to implement the constructors and the destructor?

```
class buckets{
    unsigned int *buckets;//dynamic buckets initially empty to be loaded
    unsigned int firstLoadedBucket;//the index of first loaded bucket
    unsigned int bucketLoadedNum;//the number of loaded bucket
    unsigned int loadNum;//the amount of load
    double loadFactor;//(loadNum/MAX_SIZE)
    double maxLoadFactor;//upper bound of loadFactor
  public:
    buckets(unsigned int size=MAX_SIZE, double max_factor=2);
    buckets(unsigned int load; unsigned int index;
            unsigned int size=MAX_SIZE, double max_factor=2);
    ~buckets();
    ...
```