Notes 2

Thursday, September 10, 2020 12:51 PM

Makefile:

- Comments
- Rules
- Dependency lines
- Shell lines
- Macros
- Inference Rules

Templates:

code example:

```
template <typename T, typename T2>
T2 Subtract_num(T a, T2 b)
{
  return a-b;
}

int main()
{
  int j=7;
  double i=15.0;
  cout << Subtract_num(i,j);
  return 0;
}
```

Abstraction:

- Separates purpose of a module from its implementation
  "Black Box"
- Abstraction Types:
  - Functional (Procedural)
  - Data

* Functional Abstraction
- Separating the purpose of a module from its implementation
  - Car/Engine example

* Data Abstraction
- What the operations do w the collection of data
- Know what operations can be performed, but do not know how the data is stored or how the operations are performed

Ask - What, not how
treat them as black box

Pointers:

- nullptr
  - allows you to specify that a pointer is essentially pointing at nothing.
    #include <cstddef>
  - if we initialize a pointer and then null it we have a memory leak. Why?

Ternary operator:
"?" - combines if & else in one line

example:

```
a < b ? b : a;
```

if → then → else

The stack is limited

Heap is dynamic

Some things are better on the heap.

Pointers:

points to a place in memory

```
int *a[ ];
std::cout<< *a;
```


```
Polygon * p1 = new Rect();
        ⋮
auto p2 = p1;
```

Auto Keyword

Forces compiler to declare type.

Data Members

- Members should always be private.
  - Prevents unwanted access
  - Create getters/setters in order to access them.

Hint: node should have a value in the constructor (Should be private).

2 pieces of ADT:
- operations/methods
- attributes/data

Every function you write this semester should follow this format -
- Comment/Description
- Precondition
- Postcondition

Array-Based Implementation:

- arrays are homo
- arrays are a fixed size
- arrays are of one type

- Fixed-Size Array
  - Fixed maximum Size

Valgrind: tool to check for memory leaks

C++ Review: Abstract Data Types

- The ADT Lifecycle:
  1. **Construction:** The ADT is created.
     - Locally on the stack.
     - Dynamically on the heap.
  2. **Initialization**: The variables of the ADT are set to their initial value.
     - Set the ADT's initial state.
  3. **Usage**: The ADT is now ready for use in program.
     - Invokes its member functions.
  4. **Destruction**: The ADT and it's memory is no longer usable.
     - Removed from the local stack.
     - Memory released back to heap.

C++ Review: Type Inference

- How do we allow for type inference in C++ templates?
  - C++11 Standard
- `Auto` Keyword
  - If the compiler can infer the type of a variable at the point of declaration you can use `auto` instead of the type name.
    - `int x = 4;` VS. `auto x = 4;`
- Benefit:
  - Cleaner, more readable, code.
- Consequence:
  - More readable?

Vectors:

- take the best of both worlds
- No need to manage our memory
- Vector is dynamically sized
- We can treat it like an array

Vectors

- We can even access our vector in a similar syntax as if it were an array in C++:

```
v1[10];
```

- The [] operator is present on the vector Class.
  - It also provides the method `at()` which provides the same functionality with the additional exception thrown if it is out of range (`std::out_of_range`)

Valgrind: tool to check for memory leaks

Valgrind test: type "Valgrind" space "--log-file=" "valgrind.txt" Ex1.ex

What a memory leak looks like:

Check Valgrind.txt
look for
no leaks are possible

B
D
D
B

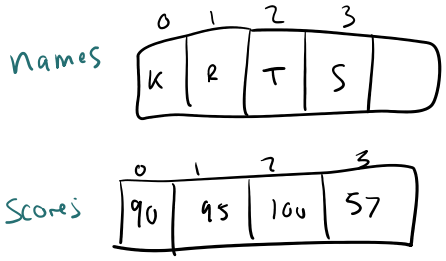
Delete

- The [] operator is present on the vector Class.
  - It also provides the method `at()` which provides the same functionality with the additional exception thrown if it is out of range (`std::out_of_range`)

Vectors

- In order to accomplish our goal of storing both the names and the scores we will need to create two different vectors.
  - Remember vectors are a templated class.

```
std::vector<std::string> names;
std::vector<double> scores;
```



names.at(0);
scores.at(0);
names.push_back("D")

Bitwise Operations

```
#include <iostream>

int main()
{
	unsigned int x = 15;
	unsigned int y = 87;

	int z(0);

	z = x & y;

	std::cout << "Bitwise AND: " << z << std::endl ;

	return 0;
}
```

Bitwise Operations

- Left Shift Operator (<<)
  - Shifts the bits to the left by the number of positions specified by expression.
- Right Shift Operator (>>)
  - Shifts the bits to the right by the number of positions specified by expression.