Last time C++

- values are only data
- types decide which function gets called when manipulating data
- function are used to group code into smaller reusable pieces
- essentially everything is a function
- two functions can have the same name if they take different parameters, this is called function overloading
- operators are also essentially functions but with a shorter calling syntax
- shorter operator syntax introduces ambiguities
- operator precedence and order of evaluation resolves ambiguities

```
int i{4}, j{5};
i = i + j > 0;
```

Fun with structs

```
struct MyStruct {
    int data_;
    void f() {
    }
    int operator ()(int i) {
        return i + data_;
    }
};
    auto s = MyStruct { 4 };
    auto o = s(1);
```

Function templates

```
int add(int l, int r)
{
    return l + r;
}

template<typename T>
T add(T l, T r)
{
    return l + r;
}

auto f = add(3.1, 2.8);
```

A less buggy string

```
#include <cstring>
#include <algorithm>
struct MyString
{
    int size_;
    char* buffer_;
    MyString() : buffer_ {}{}
    MyString(const char* in) :
        size_ {strlen(in)}, buffer_{new char[size_]}
    {
        std::copy_n(in, size_, buffer_);
    }
};
```

Using our better string

```
int main(int argc, char *argv[])
{
    auto fs = MyString{};
    auto ms = MyString{"blahhhhhhhhhhhhhhhhhhhblub"};
    auto i = sizeof(fs);
    auto j = sizeof(ms);
    return 0;
}
```

AHHHHHHH bugs!!!

Stack vs. Heap

- Size of stack objects must be known at compile time
- Stack objects lifetimes are bound to scope*
- Objects on the stack have good cache locality

- Heap objects can be any size and size need only be known at run time
- Live from when they are created with new until they are deleted
- Can be optional
- Usually have bad cache locality

Fixing our FAILs

```
struct MyString
{
    int size_;
    char* buffer_;
    MyString() : buffer_ {nullptr}{}
    MyString(const char* in) : size_ {strlen(in)},
buffer_{new char[size_]} {
        std::copy_n(in, size_, buffer_);
    }
    ~MyString()
    {
        delete[] buffer_;
    }
};
```

new and delete are like a chain saw, use safety gear!

```
int main(int argc, char *argv[])
{
    auto ms = MyString{"blahhhhhhhhhhhhhhhhhhhblub"};
    delete[] ms.buffer_;
    std::cout << ms.buffer_ << std::endl;
    return 0;
}
```

ê→ã

Did I mention C is the root of all EVIL?

```
int main(int argc, char *argv[])
{
    auto ms = MyString{"blahhhhhhhhhhhhhhhhhhhhblub"};
    std::cout << ms.buffer_ << std::endl;
    return 0;
}
```

```
blahhhhhhhhhhhhhhhhhhhhblub½½½½½½½¯■¯■¯■
```

There are a lot of smart people.... can't they be smart for me?

```
int main(int argc, char *argv[])
{
    std::string s{"blingbling"};
    std::vector<int> v{1,2,3,4};
    std::deque<int> d{5,6,7};
    std::cout << s << std::endl;
    for (auto i : v){
        std::cout << i << std::endl;
    }
    for (auto i : d){
        std::cout << i << std::endl;
    }
    return 0;
}
```

Printing is generic, lets use a template

```
template<typename T>
void printContainer(T container){
    for (auto i : container) {
        std::cout << i << std::endl;
    }
}
```

More containers

```
#include <string>
#include <vector>
#include <deque>
#include <set>
int main(int argc, char *argv[])
{
    std::set<int> s1{1,2,3,4};
    std::set<int> s2{1,2,3,4,4,4,4,4,4,4,4,4,1,2,3};
    printContainer(s1);
    printContainer(s2);
    return 0;
}
output is 12341234!
```