COP3503

Function Pointers

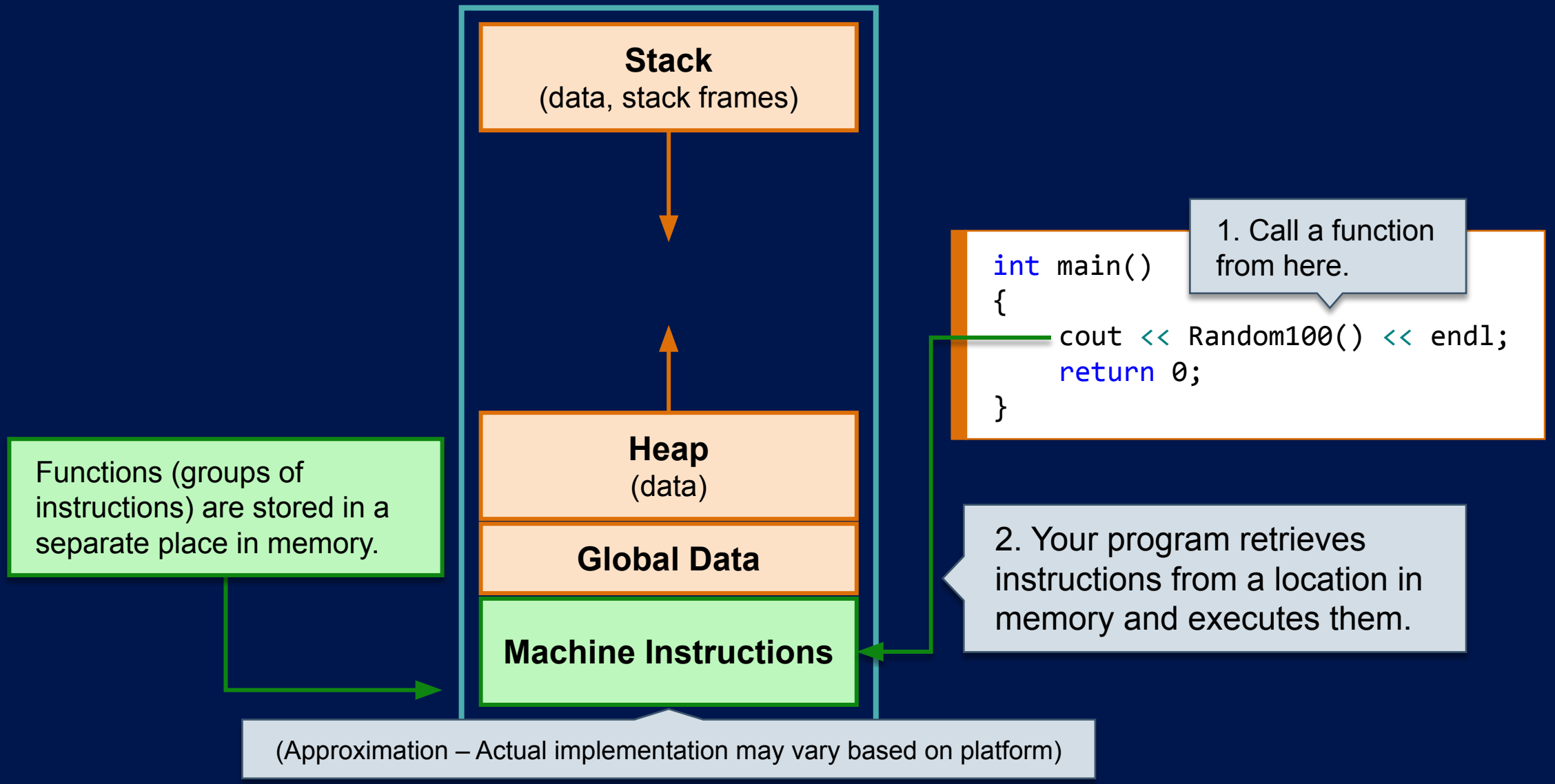
The Setup: Just Calling a Function

```
// Random100() == made up function
int main()
{
    cout << Random100() << endl;
    return 0;
}

// What if we forgot ()?
int main()
{
    cout << Random100 << endl;
    return 0;
}
```

What would this print out?

Something like 07FF6A99, a memory address.

Functions Live in Memory



Sorting Examples

- By numeric value (**ascending**, low-to-high, or **descending**, high-to-low)
- Sort a list of items for sale, by price or other attributes:
  - Physical attributes like size or weight
  - Rating, find the best (or worst!) reviewed products first
  - Time ending (if time/date is a variable, think auctions or limited-duration sales)
- Vehicles sorted by mileage
- Books sorted by publication date
- Names and strings, sorting in alphabetical order
  - Strings are made of characters, and characters are just numbers…

Functions Have Memory Addresses

- The name of a function is a pointer to its **address**.
- Calling a function retrieves and executes the instructions at that memory address.
- You can store that address with a **function pointer**.
  - Use it later, passing to other functions or classes.
  - The same reason you would use any kind of pointer.

```
int Random100()
{/* insert cool stuff here */ }

// Store a pointer to the function
// Variable name == ptr
int(*ptr)() = Random100;
```

```
// Using a function pointer
// Call it just like a function
int random = ptr();
```

But… Why not just call the original function?

More Function Pointer Examples

Function Prototype

```
int Random100();
void DoSomeStuff(int value);
int SomeFunc(float x, char y);
int SomeFunc2(float x, char y);
```

Function Pointer

```
int (*functPtr)() = Random100;
void (*p)(int) = DoSomeStuff;
int (*var)(float, char) = SomeFunc;
var = SomeFunc2;
```

You can reuse var, if you want (it's just a pointer, after all)

Assuming the return type and parameters are the same.

- In short:
  - Copy the function prototype and
    1. Replace the function name with □ (*YOUR_VARIABLE_NAME)
    2. Remove the names of any parameters `(int a, float b)` □ `(int a, float b)`
- The syntax is kind of gross, but… it's the way these things work.
- There are other C++ features that can help alleviate this.

```
auto functionPointer = DoSomeStuff; // #thanksauto
```

Passing Function Pointers

```
void SortStuff(vector<int>& numbers) {
    // if numbers[i] > numbers[i+1]
        // Swap numbers[i] and numbers[i+1]
}


// Need to sort another way? Create two versions...
void SortStuffAscending(vector<int>& numbers) {
    // if numbers[i] > numbers[i+1], swap...
}

void SortStuffDescending(vector<int>& numbers) {
    // if numbers[i] < numbers[i+1], swap...
}
```

This function is locked into sorting one way (ascending or descending).

Not the most ideal way to write code...

Alternative: Pull the Comparison into a Function

```
void SortStuff(vector<int>& numbers)
{
    // if (SomeComparison(numbers[i], numbers[i + 1]))
    // Swap two values
}

// Pass a function pointer!
void SortStuff(vector<int>& numbers, bool(*SomeComparison)(int, int))
{
    if (SomeComparison(numbers[i], numbers[i + 1]))
    // Swap two values
}
```

If **some** comparison is true, then do something—we just don't define what that comparison is.

So… where does this come from?

Now, to sort something we need:

1. The stuff to sort.
2. A pointer to a function which compares two integers (in some unknown way), and returns a bool.

Writing Functions to Pass to Functions

```
void SortStuff(vector<int>& numbers, bool(*compare)(int, int));

// Create two varieties of comparisons
bool Ascending(int a, int b)
{
    return a > b;
}
bool Descending(int a, int b)
{
    return a < b;
}

// One function, two sorting options!
SortStuff(someVector, Ascending);
SortStuff(someVector, Descending);
```

The functions (and the code inside them) can be sent as data.

This lets us build customizable sets of instructions!

Array of Function Pointers

```
// Function prototypes
double Add     (double a, double b);
double Subtract(double a, double b);
double Multiply(double a, double b);
double Divide  (double a, double b);

// One function pointer
double(*ptr)(double, double) = Add;

// Array of function pointers
double(*operations[4])(double, double);
operations[0] = Add;
operations[1] = Subtract;
operations[2] = Multiply;
operations[3] = Divide;
```

Brackets and the array size go after the variable name, just like you would anywhere else.

```
// Use an enum to identify elements
enum OPS {ADD, SUBTRACT, MULTIPLY, DIVIDE};
```

Using the Array of Function Pointers

```
// Array of function pointers
double(*operations[4])(double, double);
operations[0] = Add;
operations[1] = Subtract;
operations[2] = Multiply;
operations[3] = Divide;
```

```
cout << "Select an operation:\n";
cout << "1. Add\n";
cout << "2. Subtract\n";
cout << "3. Multiply\n";
cout << "4. Divide\n";

double opIndex, value1, value2;
cin >> opIndex;

cout << "Enter two values:\n";
cin >> value1;
cin >> value2;
```

```
double result = operations[opIndex-1](value1, value2);
```

This one line works for any number of operations!

Alternatives to Arrays of Functions

What if you add or remove an operation?

```
double result = operations[opIndex-1](value1, value2);
```

This one line of code doesn't have to change!

— OR —

```
double result;

if (opIndex == 1)
    result = Add(value1, value2);
else if (opIndex == 2)
    result = Subtract(value1, value2);
else if (opIndex == 3)
    result = Multiply(value1, value2);
else if (opIndex == 4)
    result = Divide(value1, value2);
```

These cases have to be modified to reflect all the options.

```
switch (opIndex)
{
case 1:
    result = Add(value1, value2);
    break;
case 2:
    result = Subtract(value1, value2);
    break;
case 3:
    result = Multiply(value1, value2);
    break;
case 4:
    result = Divide(value1, value2);
    eak;
```

Not impossible, but still "one more thing" to do…

Use using to Clean Up Some Ugliness

```
bool Ascending(int a, int b)
{
    return a > b;
}

// Old way to do it
bool(*funcPtr)(int, int) = Ascending;

// New way to do it...
// Comparison is now an alias (a type)
using Comparison = bool(*)(int, int);
Comparison myPtr = Ascending;// #muchbetter
```

You may see this a lot in C++ code written by someone else.

Take some core part of the language and relabel it so that humans can work with it more easily.

Use using to Clean Up Some Ugliness

```
// This ugliness…
void SortStuff(vector<int>& numbers, bool(*compare)(int, int));

// Turns into this!
void SortStuff(vector<int>& numbers, Comparison c);

// Arrays and containers
Comparison funcs[3];
funcs[0] = SomeFunction;
funcs[1] = nullptr;

vector<Comparison> comparers;
map<string, Comparison> operations;
operations["Add"] = Add;
```

They're all still using function pointers, just by another name.

A little bit of cleaning up our code at the source can go a long way for the rest of it.

Templates Are an Alternative Too

```
// Nice, clean alias
void SortStuff(vector<int>& numbers, Comparison c)
{
    if (c(numbers[i], numbers[i + 1]))
    { // swap, or something else… }
}

// Templates can work as well
template <typename T>
void SortStuff(vector<int>& numbers, T compareFunction)
{
    if (compareFunction(numbers[i], numbers[i + 1]))
    { // swap, or something else… }
}
```

```
// T == bool(*)(int, int)
SortStuff(numbers, Ascending);
```

This template says “I assume the thing you pass me is a function pointer. I’m going to try and use it like a function.”

You’ll likely see one of these two options more than “raw” function pointers.

Storing Function Pointers in Classes

```
class FunctionHolder
{
     // Function pointer to store a single function
     void (*_singleAction)();

     // Container for grouping multiple function calls together
     vector<void(*)()> _actions;
public:
     void AddAction(void (*a)());// Add a pointer to the vector
     void DoAllActions();             // Call all stored functions
};
```

```
void FunctionHolder::AddAction(void (*a)())
{
     _actions.push_back(a);
}
```

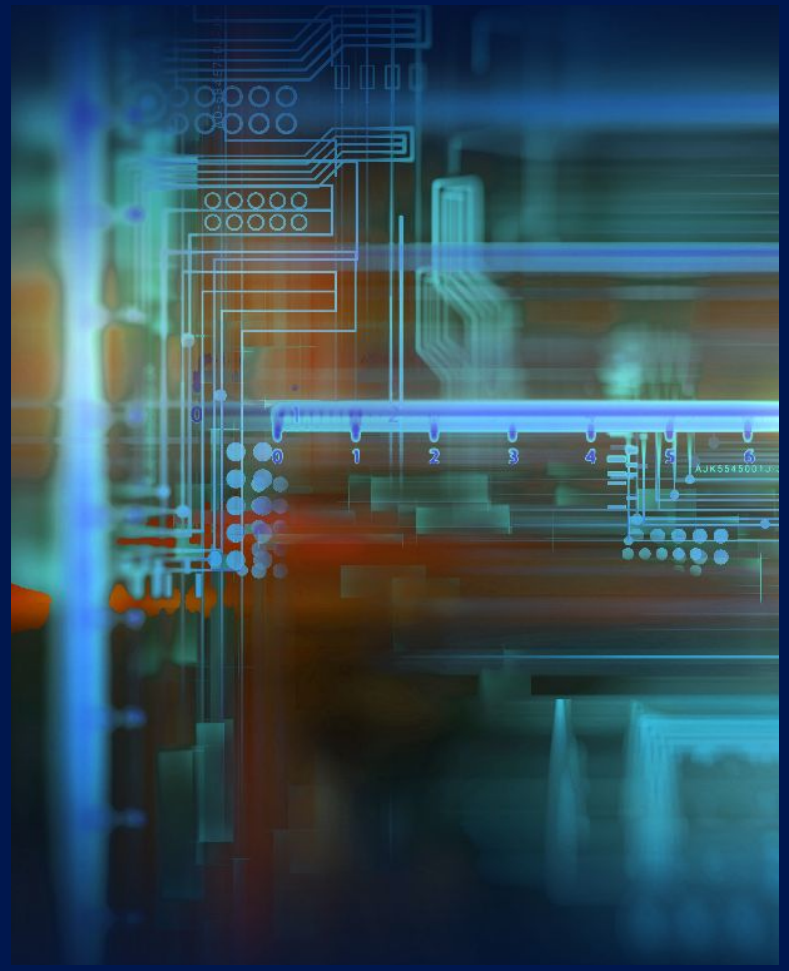
```
void FunctionHolder::DoAllActions()
{
     for (unsigned int i = 0; i < _actions.size(); i++)
          _actions[i]();
}
```

```
FunctionHolder obj;
obj.AddAction(Foo);
obj.AddAction(Bar);
obj.AddAction(Baz);

// Execute all stored functions
obj.DoAllActions();

SomeOtherFunction(obj);
```

Recap

- **Functions have memory addresses** like our variables.
  - **Function pointers** let us store those addresses for later use.
- Function pointers allow you **use functions as data**.
- All operations that apply to data can now apply to functions:
  - Storing functions in containers
  - Passing functions to functions
  - Storing pointers to functions for later use
- They are very effective when building systems that allow for combinations of operations.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.