CSCI 200: Foundational Programming Concepts & Design

Memory: Stack & The Free Store

Pointers

Learning Outcomes For Today

- Explain the difference between the stack & the free store and the contents of each.
- Implement & manipulate pointers to reference memory on the stack or the free store.

Important Note

- We're going to be using pointers extensively here on out.

- If anything's unclear, ask!

On Tap For Today

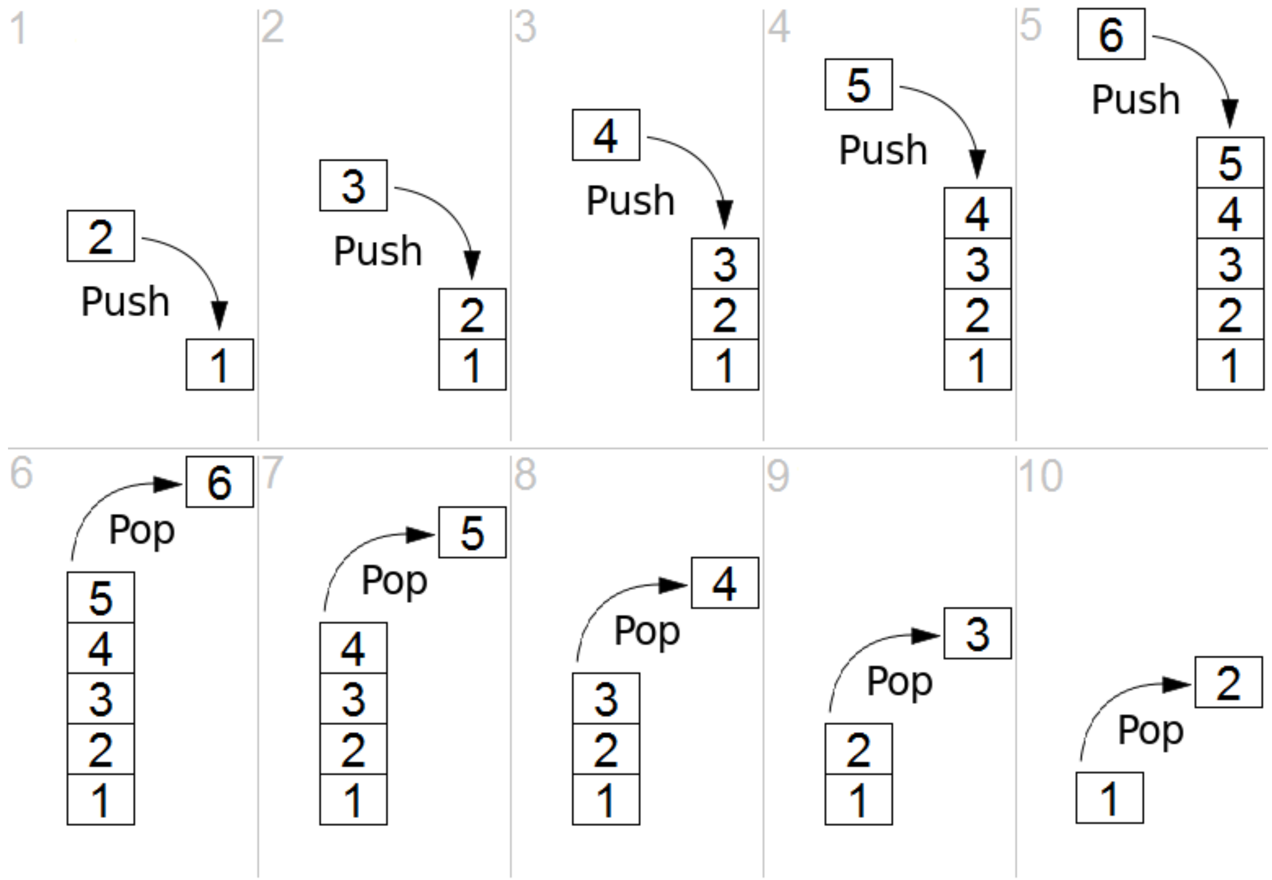
- The Stack
- The Free Store
- Pointers

On Tap For Today

- The Stack
- The Free Store
- Pointers

A Stack

- Last In First Out (LIFO)



Stack Frame

- One aspect of where scope comes from
- A stack frame contains
  - Current point of execution
  - Local variables
  - Function parameters
- New function call?
  - Push a new stack frame
- Function returns?
  - Pop the top stack frame

The Call Stack

```
int sum(int x, int y) {
    return x + y;
}

int main() {
    int x = 2;
    int y = 5;
    int z = sum(x, y);
}
```

Address	Value	Stack
0x40960014		
0x40960018		
0x4096001c		
0x40960020		
0x40960024		
0x40960028		
0x4096002c		
0x40960030		
0x40960034		
0x40960038		
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018		
0x4096001c		
0x40960020		
0x40960024		
0x40960028		
0x4096002c		
0x40960030		main() : 05
0x40960034		
0x40960038		
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018		
0x4096001c		
0x40960020		
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		`main()` : 06 (int) x 0x40960028
0x40960034		
0x40960038		
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018	5	
0x4096001c		
0x40960020		
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		`main()` : 07 (int) x 0x40960028 (int) y 0x40960018
0x40960034		
0x40960038		
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018	5	
0x4096001c		
0x40960020		
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		main() : 08 (int) x 0x40960028 (int) y 0x40960018 (int) z 0x40960034
0x40960034		
0x40960038		
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

<table>
<tr><th>Address</th><th>Value</th><th>Stack</th></tr>
<tr><td>0x40960014</td><td></td><td rowspan="3"></td></tr>
<tr><td>0x40960018</td><td>5</td></tr>
<tr><td>0x4096001c</td><td></td></tr>
<tr><td>0x40960020</td><td>2</td><td rowspan="4">sum() : 01
(int) x 0x40960020
(int) y 0x40960038</td></tr>
<tr><td>0x40960024</td><td></td></tr>
<tr><td>0x40960028</td><td>2</td></tr>
<tr><td>0x4096002c</td><td></td></tr>
<tr><td>0x40960030</td><td></td><td rowspan="4">main() : 08
(int) x 0x40960028
(int) y 0x40960018
(int) z 0x40960034</td></tr>
<tr><td>0x40960034</td><td></td></tr>
<tr><td>0x40960038</td><td>5</td></tr>
<tr><td>0x4096003c</td><td></td></tr>
</table>

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018	5	
0x4096001c		
0x40960020	2	sum() : 02 (int) x 0x40960020 (int) y 0x40960038
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		main() : 08 (int) x 0x40960028 (int) y 0x40960018 (int) z 0x40960034
0x40960034		
0x40960038	5	
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018	5	
0x4096001c		
0x40960020	2	
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		`main()` : 08 (int) x 0x40960028 (int) y 0x40960018 (int) z 0x40960034
0x40960034	7	
0x40960038	5	
0x4096003c		

The Call Stack

```
01 int sum(int x, int y) {
02     return x + y;
03 }
04
05 int main() {
06     int x = 2;
07     int y = 5;
08     int z = sum(x, y);
09 }
```

Address	Value	Stack
0x40960014		
0x40960018	5	
0x4096001c		
0x40960020	2	
0x40960024		
0x40960028	2	
0x4096002c		
0x40960030		main() : 09 (int) x 0x40960028 (int) y 0x40960018 (int) z 0x40960034
0x40960034	7	
0x40960038	5	
0x4096003c		

Stack Problems

1. Reference to local variables for a function are deleted
   – May want to keep those variables after the function returns

Stack Problems

1. Reference to local variables for a function are deleted
2. Stack has a limited size. Storing too much causes "stack overflow"
   - On Win (~1 MB default):

```
double arr[100000];   // success (0.1M doubles)
double arr[1000000];  // failure (1M doubles)
```

   - On OS X (~8 MB default):

```
double arr[1000000];  // success (1M doubles)
double arr[10000000]; // failure (10M doubles)
```

Stack Problems

1. Reference to local variables for a function are deleted
2. Stack has a limited size. Storing too much causes "stack overflow"
   - On Win (~1 MB default):
     - `~64K stack frames with one` **`double`** `each`
   - On OS X (~8 MB default):
     - `~261K stack frames with one` **`double`** `each`

Stack Problems

1. Reference to local variables for a function are deleted
2. Stack has a limited size. Storing too much causes "stack overflow"
3. In many cases, we don't know how much memory we'll need at compile time.
   - Need to allocate memory dynamically at run-time.

Need Another Place

- Enter…the Free Store

On Tap For Today

- The Stack
- The Free Store
- Pointers

The Free Store

- A pool of unused memory for your program to use
  - Analogous to the “Heap” in C
  - Free Store specific to C++

Stack vs. Free Store

Stack	Free Store
Organized	Unorganized
Efficient	Less efficient (but we don't care)
Accessed "directly"	
Storage of variables known in advance	Dynamic Storage at run time
And more!	And more!

When to use the Free Store?

1. When you need memory to persist beyond function scope

2. When you have a large amount of data to store in memory
   - Stack size is smaller than the free store size

3. When you need a resizable / dynamic data structure.
   - We cannot resize data structures on the stack.

Free Store Location

- Exact memory location not known in advance

- How does your code access data if location isn't known in advance?
  - Pointers!

On Tap For Today

- The Stack
- The Free Store
- Pointers

What is a pointer?

- A variable that "stores" a memory address

What is printed?

```
int a = 1;
cout << a << endl;
cout << &a << endl;
```

```
1
0x6f304018
```

Identifier	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1
	0x6fe0401c	

- & - reference operator

What is printed?

```
int a = 1;
int b = 7;
int c = 12;
```

Variable	Memory Address	Value
b	0x6fe04014	7
a	0x6fe04018	1
c	0x6fe0401c	12

```
cout << &a << endl; //print the address of a
cout << &b << endl; //print the address of b
cout << &c << endl; //print the address of c
```

Addresses are numbers

```
double a = 1.0;

unsigned int pA = &a;
```

- **`pA`** stores the address of **`a`**

Variable	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1.0
pA	0x6fe0401c	0x6fe04018

Addresses are numbers

```c
double a = 1.0;
unsigned int pA = &a;
```

- **`pA`** stores the address of **`a`**

Variable	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1.0
pA	0x6fe0401c	0x6fe04018

- But we usually care more about the value at the address, not the address itself

Addresses are numbers

```
double a = 1.0;

unsigned int pA = &a;
```

Variable	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1.0
pA	0x6fe0401c	0x6fe04018

- **`pA`** stores the address of **`a`**

- If we want to use the value whose address is **`pA`**, we have to know its type and cast (and do the actual lookup)

Pointer Syntax

```
double a = 1.0;

double *pA = &a;
```

- **`pA`** stores the address of a double **`a`**

Variable	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1.0
pA	0x6fe0401c	0x6fe04018

- Using pointer syntax, no casting involved

Pointer Type

```
// Primitive Types

bool
char
int
float
double

void*  // T*

double a = 1.0;
double *pA = &a;
```

Dereferencing a Pointer

- Use the indirection operator to work with the memory location being pointed to

```
double a = 1.0;
double *pA = &a;


cout << pA << endl;
cout << *pA << endl;
```

Variable	Memory Address	Value
	0x6fe04014	
a	0x6fe04018	1.0
pA	0x6fe0401c	0x6fe04018

Practice: What is printed?

```
double a = 1.0;
double *pA = &a;
a = 18;
*pA = 22;
cout << a << endl;
```

```
22
```

What's the difference?

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

int main() {
  int a = 7, b = 7;
  int *pA = &a, *pB = &b;
  if( pA == pB )
    cout << "what is this testing?" << endl;
  if( *pA == *pB )
    cout << "and what is this testing?" << endl;
  return 0;
}
```

What’s the difference?

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

int main() {
  int a = 7, b = 7;
  int *pA = &a, *pB = &b;
  if( pA == pB )     // points to same object
    cout << "what is this testing?" << endl;
  if( *pA == *pB )   // points to equivalent objects
    cout << "and what is this testing?" << endl;
  return 0;
}
```

Storing Objects on the Free Store

- Use a pointer!

```
int *pNumCars = new int;
```

- * - indirection operator

- **new** – "Computer, allocate enough memory in the free store for one object and tell me the starting address where the object will be stored."

`new`

- **`new`** returns a pointer

```
int *pNumCars = new int;
```

- **`pNumCars`** is a pointer to an integer variable on the free store

`new` and `delete`

- **`new`**: allocates memory on the free store
- **`delete`**: returns used memory to the free store

```
int *pNumCars = new int;

delete pNumCars;
```

- Very common use with dynamic arrays

Precedence Table

<table>
<tr><th>Precedence</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>1</td><td>Parenthesis:
( )</td><td>Innermost First</td></tr>
<tr><td>2</td><td>Postfix Unary Operators:
a++ a-- f()</td><td>Left to Right</td></tr>
<tr><td>3</td><td>Prefix Unary Operators:
++a --a +a -a !a (type)a &a *p new delete</td><td>Right to Left</td></tr>
<tr><td>4</td><td>Binary Operators:
a*b a/b a%b</td><td rowspan="6">Left to Right</td></tr>
<tr><td>5</td><td>Binary Operators:
a+b a-b</td></tr>
<tr><td>6</td><td>Relational Operators:
a<b a>b a<=b a>=b</td></tr>
<tr><td>7</td><td>Relational Operators:
a==b a!=b</td></tr>
<tr><td>8</td><td>Logical Operators:
a&&b</td></tr>
<tr><td>9</td><td>Logical Operators:
a||b</td></tr>
<tr><td>10</td><td>Assignment Operators:
a=b a+=b a-=b a*=b a/=b a%=b</td><td>Right to Left</td></tr>
</table>

The Free Store

- A pool of unused memory for your program to use
  - Analogous to the "Heap" in C
  - Free Store specific to C++
- FYI for reference and to distinguish

Language	Name of Memory Area	How to Allocate Memory	How to Deallocate Memory
C	Heap	`malloc()/calloc()`	`free()`
C++	Free Store	`new`	`delete`

  - Do not mix the two styles when working with pointers!
    A pointer can only interface with one pair of commands