Postfix Notation and the Call Stack

Stacks

Outline

- Postfix
- The call stack
- Stacks

Postfix

And Stacks

Reverse Polish Notation

- Reverse Polish Notation (RPN)
  - Also known as postfix notation
  - A mathematical notation
    - Where every operator follows its operands
  - Invented by Jan Łukasiewicz in 1920
- Example
  - Infix: 5 + ((1 + 2) * 4) – 3
  - RPN: 5 1 2 + 4 * + 3 –

RPN Example

To evaluate postfix expressions read from left to right

5 1 2 + 4 * + 3 −

store 5

store 1

store 2

Apply + to the last two operands

1 + 2

```
for each input symbol
   if symbol is operand
      store(operand)
   if symbol is operator
      RHS = remove()
      LHS = remove()
      result =
         LHS operator RHS
      store(result)
result = remove()
```

2

1

5

RPN Example

To evaluate postfix expressions read from left to right

5 1 2 + 4 * + 3 −

store 5

store 1

store 2

Apply + to the last two operands

1 + 2

store 3

store 4

Apply * to the last two operands

3 * 4

4
3
5

```
for each input symbol
   if symbol is operand
      store(operand)
   if symbol is operator
      RHS = remove()
      LHS = remove()
      result =
         LHS operator RHS
      store(result)
result = remove()
```

RPN Example

5 1 2 + 4 * + 3 −

To evaluate postfix expressions read from left to right

store 5

store 1

store 2

Apply + to the last two operands

1 + 2

store 3

store 4

Apply * to the last two operands

3 * 4

store 12

Apply + to the last two operands

5 + 12

12
5

```
for each input symbol
   if symbol is operand
      store(operand)
   if symbol is operator
      RHS = remove()
      LHS = remove()
      result =
         LHS operator RHS
      store(result)
result = remove()
```

RPN Example

To evaluate postfix expressions read from left to right

5 1 2 + 4 * + 3 −

store 5 | store 1 | store 2

Apply + to the last two operands | 1 + 2

store 3 | store 4

Apply * to the last two operands | 3 * 4

store 12

Apply + to the last two operands | 5 + 12

store 17 | store 3

Apply - to the last two operands | 17 - 3

3

17

```
for each input symbol
    if symbol is operand
        store(operand)
    if symbol is operator
        RHS = remove()
        LHS = remove()
        result =
            LHS operator RHS
        store(result)
result = remove()
```

RPN Example

5 1 2 + 4 * + 3 −

To evaluate postfix expressions read from left to right

store 5 | store 1 | store 2

Apply + to the last two operands | 1 + 2

store 3 | store 4

Apply * to the last two operands | 3 * 4

store 12

Apply + to the last two operands | 5 + 12

store 17 | store 3

Apply - to the last two operands | 17 - 3

store 14

14

retrieve answer | 14

```
for each input symbol
   if symbol is operand
      store(operand)
   if symbol is operator
      RHS = remove()
      LHS = remove()
      result =
         LHS operator RHS
      store(result)
result = remove()
```

Describing a Data Structure

- What are the storage properties of the data structure that was used?
  - How were the operands stored and removed?
- Operands were never inserted between other operands
  - The last item to be entered was always the first item to be removed
  - Known as LIFO (Last In First Out)
- This data structure is known to as a *stack*

The Call Stack

Another Stack Example

Functions

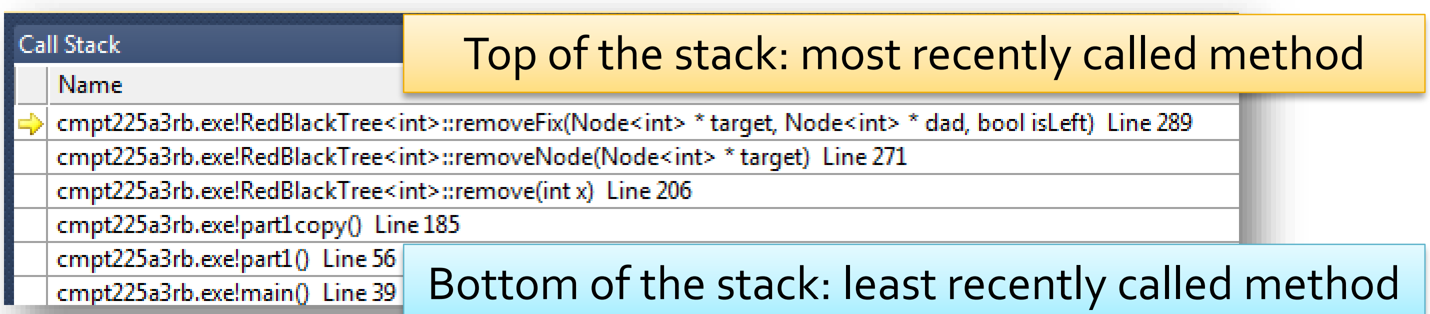
- Programs typically involve more than one function call and contain
  - A *main* function
  - Which calls other functions as required

```
int main()
{
    foo();
    bar();
    // …
}
```

- Each function requires space in main memory for its variables and parameters
  - This space must be allocated and de-allocated in some organized way

Organizing Function Calls

- Many programming languages use a *call stack* to implement function calling
  - When a method is called, its line number and other data are *pushed* onto the call stack
  - When a method terminates, it is *popped* from the call stack
  - Execution restarts at the indicated line number in the method currently at the top of the stack



Stack Frames

- Information stored on the call stack about a function is itself stored in a *stack frame*
  - Sometimes referred to as an *activation record*
- Stack frames store
  - The arguments passed to the function
  - The return address back to the calling function
  - Space for the function's local variables
- Stack memory is allocated and de-allocated without explicit instructions from a programmer
  - And is therefore referred to as *automatic* storage

Call Stack and Memory

- When a function is called space is allocated for it on the call stack
  - This space is allocated *sequentially*
- Once a function terminates the memory it used is no longer required
  - And becomes available for the next function call
- Execution returns to the previous function
  - Which is now at the top of the call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
void squareArray(int a[], int n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	5
exp	2
result	1
i	1
squareArray	
a	aff02b5c
n	2
i	0
x	5
main	
n	2
arr	5 17
sum	-

call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
void squareArray(int a[], int n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	5
exp	2
result	25
i	3
squareArray	
a	aff02b5c
n	2
i	0
x	5
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
      int n = 2;
      double arr[] = {5,17};
      squareArray(arr, n);
      int sum = sumArray(arr, n);
      cout << sum << endl;
      return 0;
}
```

```
 void squareArray(int a[], int n){
      for(int i=0; i < n; i++){
            int x = a[i];
            a[i] = power(x, 2);
      }
}
```

```
double power(double x, int exp){
      double result = 1;
      for(int i=1; i <= exp; i++){
            result *= x;
      }
      return result;
}
```

```
double sumArray(double a[], int n){
      double sum = 0;
      for(int i=0; i < n; i++){
            sum += a[i];
      }
      return sum;
}
```

squareArray	
a	aff02b5c
n	2
i	1
x	17
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
void squareArray(int a[], int n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	17
exp	2
result	289
i	3
squareArray	
a	aff02b5c
n	2
i	2
x	17
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
      int n = 2;
      double arr[] = {5,17};
      squareArray(arr, n);
      int sum = sumArray(arr, n);
      cout << sum << endl;
      return 0;
}
```

```
double sumArray(double a[], int n){
      double sum = 0;
      for(int i=0; i < n; i++){
            sum += a[i];
      }
      return sum;
}
```

```
 void squareArray(int a[], int n){
      for(int i=0; i < n; i++){
            int x = a[i];
            a[i] = power(x, 2);
      }
}
```

```
double power(double x, int exp){
      double result = 1;
      for(int i=1; i <= exp; i++){
            result *= x;
      }
      return result;
}
```

sumArray	
a	aff02b5c
n	2
sum	314
i	2
main	
n	2
arr	25 289
sum	314

call stack

Returning Values

- In the example, functions returned values assigned to variables in other functions
  - They did not affect the *amount of memory* required by previously called functions
  - That is, functions *below* them on the call stack
- Stack memory is sequentially allocated
  - It is not possible to increase memory assigned to a function previously pushed onto the stack

Stacks

Postfix and Stacks

- A stack is a natural choice to store data for postfix notation arithmetic or the call stack
  - Operands or stack frames are stored at the top
    - And removed from the top
- Notice that we have not (yet) discussed how a stack should be implemented
  - Just *what* it does
- An example of an *Abstract Data Type*

Stacks

- A stack only allows items to be inserted and removed at *one end*
  - The *top* of the stack
- Access to other items is not allowed

Stack Operations

- A stack should implement at least the first two of these operations
  - *push* – insert an item at the top of the stack
  - *pop* – remove and return the top item
  - *peek* – return the top item
- The operations should be performed efficiently
  - The definition of efficiency varies between ADTs
  - Note that the order of the items in a stack is based solely on the order in which they arrive

A Design Note

- Assume that we plan on using a stack that will store integers and have these methods
  - `void push(int)`
  - `int pop()`
- We can design other modules that use these methods
  - Without having to know anything about how they, or the stack itself, are implemented

Classes

- We will use classes to encapsulate stacks
  - Encapsulate – enclose in
- A class is a programming construct that contains
  - Data for the class, and
  - Operations of the class
  - More about classes later …

Implementing a Stack

- The stack ADT can be implemented using a variety of data structures, e.g.
  - Arrays
  - Linked Lists
- Both implementations must implement all the stack operations
  - It is expected that *push* and *pop* run in constant time
    - Time that is independent of the number of items in the stack