

Lecture No.08

Data Structures

# Converting Infix to Postfix

- Example:  $(A + B) * C$

symb	postfix	stack
(		(
A	A	(
+	A	( +
B	AB	( +
)	AB +	
*	AB +	*
C	AB + C	*
	AB + C *	

# C++ Templates

- We need a stack of operands and a stack of operators.
- Operands can be integers and floating point numbers, even variables.
- Operators are single characters.
- We would have to create classes FloatStack and CharStack.
- Yet the internal workings of both classes is the same.

# C++ Templates

- We can use C++ Templates to create a “template” of a stack class.
- Instantiate float stack, char stack, or stack for any type of element we want.

# Stack using templates

Stack.h:

---

```
template <class T>
class Stack {
public:
    Stack();
    int empty(void); // 1=true, 0=false
    int push(T &); // 1=successful, 0=stack overflow
    T pop(void);
    T peek(void);
    ~Stack();
private:
    int top;
    T* nodes;
};
```

# Stack using templates

## Stack.cpp

---

```
#include <iostream.h>
#include <stdlib.h>
#include "Stack.cpp"

#define MAXSTACKSIZE 50

template <class T>
Stack<T>::Stack()
{
    top = -1;
    nodes = new T[MAXSTACKSIZE];
}
```

# Stack using templates

## Stack.cpp

---

```
template <class T>
Stack<T>::~~Stack()
{
    delete nodes;
}

template <class T>
int Stack<T>::empty(void)
{
    if( top < 0 ) return 1;
    return 0;
}
```

# Stack using templates

## Stack.cpp

---

```
template <class T>
int Stack<T>::push(T& x)
{
    if( top < MAXSTACKSIZE ) {
        nodes[++top] = x;
        return 1;
    }
    cout << "stack overflow in push.\n";
    return 0;
}
```



# Stack using templates

## Stack.cpp

---

```
template <class T>
T Stack<T>::pop(void)
{
    T x;
    if( !empty() ) {
        x = nodes[top--];
        return x;
    }
    cout << "stack underflow in pop.\n";
    return x;
}
```

# Stack using templates

main.cpp

---

```
#include "Stack.cpp"
int main(int argc, char *argv[]) {
    Stack<int> intstack;
    Stack<char> charstack;
    int x=10, y=20;
    char c='C', d='D';

    intstack.push(x);    intstack.push(y);
    cout << "intstack: " << intstack.pop() << ", "
         << intstack.pop() << "\n";

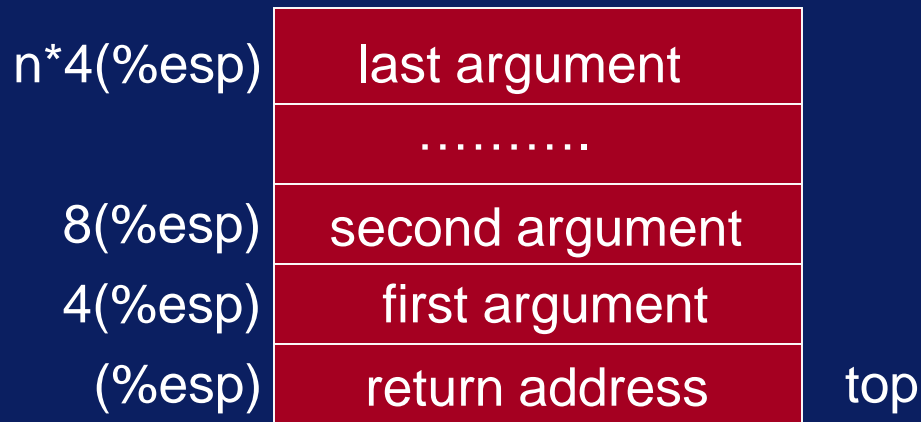
    charstack.push(c); charstack.push(d);
    cout << "charstack: " << charstack.pop() << ", "
         << charstack.pop() << "\n";
}
```

# Function Call Stack

- Stacks play a key role in implementation of function calls in programming languages.
- In C++, for example, the “call stack” is used to pass function arguments and receive return values.
- The call stack is also used for “local variables”

# Call Stack

- In GCC, a popular C/C++ compiler on Intel platform, stack entries are:



# Call Stack

Example: consider the function:

```
int i_avg (int a, int b)
{
    return (a + b) / 2;
}
```

```
# Stack layout on entry:
#
# 8(%esp) b
# 4(%esp) a
# (%esp) return address
```

# Call Stack

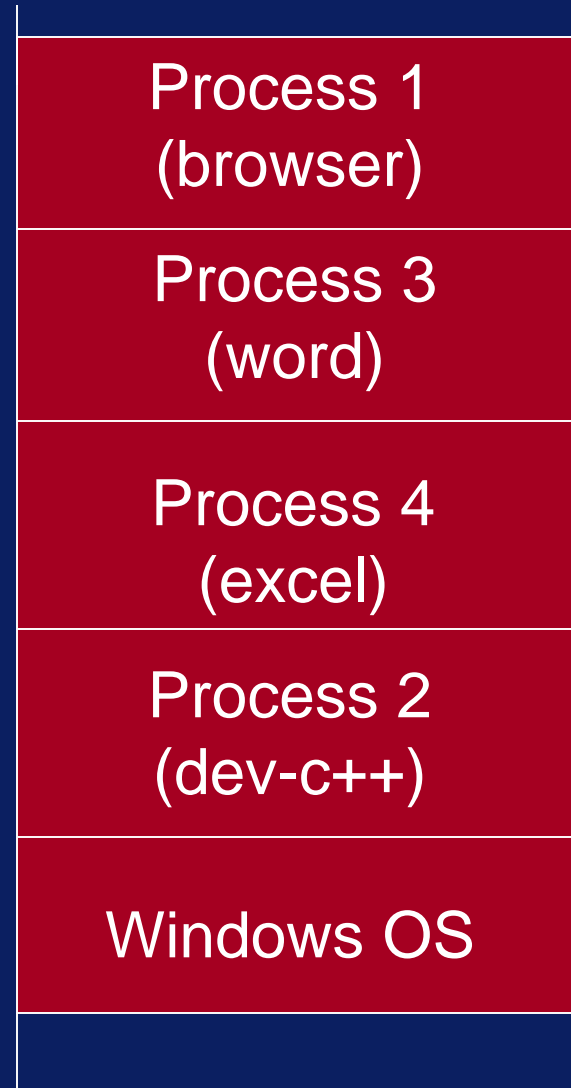
Example: consider the function:

```
int i_avg (int a, int b)
{
    return (a + b) / 2;
}
```

```
.globl _i_avg
_i_avg:
    movl 4(%esp), %eax
    addl 8(%esp), %eax # Add the args
    sarl $1, %eax # Divide by 2
    ret # Return value is in %eax
```

# Memory Organization

- When a program (.exe) is run, it is loaded in memory. It becomes a *process*.
- The process is given a block of memory.
- [Control-Alt-DEL]



# Task Manager

Windows Task Manager

File Options View Help

Applications Processes Performance

Image Name	PID	CPU	CPU Time	Mem Usage	Page Faults	I/O Reads	I/O Writes	I/O Read Bytes	I/O Write Bytes
svchost.exe	468	00	0:00:02	8,048 K	2,413	41,100	39,264	101,443,489	100,402,717
SERVICES.EXE	224	00	0:00:03	5,788 K	2,056	13,009	11,765	1,444,964	914,388
POProxy.exe	1088	00	0:00:01	5,428 K	1,453	1,860	0	883,006	0
alertsvc.exe	844	00	0:00:00	4,764 K	1,338	42	2	13,566	12
explorer.exe	888	00	0:00:20	4,696 K	16,182	8,744	127	7,060,760	2,358,902
SPOOLSV.EXE	428	00	0:00:00	3,640 K	1,336	2,095	2	1,008,896	12
svchost.exe	400	00	0:00:00	3,532 K	3,809	74	5	32,104	288
navapvc.exe	520	00	0:00:01	3,244 K	2,960	139,462	2	6,069,556	12
mstask.exe	632	00	0:00:00	3,116 K	859	48	15	19,436	356
POWERPNT.EXE	984	00	0:00:13	2,428 K	8,602	29,869	88	2,696,072	447,454
taskmgr.exe	836	01	0:00:02	2,424 K	617	0	0	0	0
CSRSS.EXE	172	00	0:00:15	1,956 K	3,037	39,254	0	981,959	0
navapw32.exe	1172	00	0:00:00	1,752 K	484	99	0	53,446	0
npssvc.exe	580	00	0:00:00	1,644 K	410	19	4	2,128	72
hidserv.exe	488	00	0:00:00	1,352 K	338	2	2	40	12
LSASS.EXE	236	00	0:00:00	868 K	2,624	1,249	807	1,469,654	137,653
regsvc.exe	616	00	0:00:00	740 K	188	2	2	54	12
WINLOGON.EXE	192	00	0:00:06	544 K	5,381	238	70	2,860,452	35,333
SMSS.EXE	144	00	0:00:00	336 K	621	236	40	14,985,728	31,744
System	8	00	0:00:27	212 K	3,160	71	1,749	33,616	1,387,471
WinMgmt.exe	684	00	0:00:10	164 K	5,911	464	23,934	5,308,962	6,611,687
System Idle Process	0	99	1:19:22	16 K	1	0	0	0	0

End Process

Processes: 22 CPU Usage: 1% Mem Usage: 88124K / 436532K



# Memory Organization

