Anatomy of a short

The range of values we can store in a short is -32768 to +32767:

<table>
<thead>
<tr>
<th>BIT PATTERN</th>
<th>DECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111111111111111</td>
<td>+32767</td>
</tr>
<tr>
<td>0111111111111110</td>
<td>+32766</td>
</tr>
<tr>
<td>...............</td>
<td>...............</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>+2</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>+1</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>1111111111111111</td>
<td>-1</td>
</tr>
<tr>
<td>1111111111111110</td>
<td>-2</td>
</tr>
<tr>
<td>...............</td>
<td>...............</td>
</tr>
<tr>
<td>1000000000000001</td>
<td>-32767</td>
</tr>
<tr>
<td>1000000000000000</td>
<td>-32768</td>
</tr>
</tbody>
</table>
Anatomy of an int

The range of values we can store in an int is -2147483648 to +2147483647:

<table>
<thead>
<tr>
<th>BIT PATTERN</th>
<th>DECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111111111111111111111111111111</td>
<td>+ 2147483647</td>
</tr>
<tr>
<td>011111111111111111111111111111110</td>
<td>+ 2147483646</td>
</tr>
<tr>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>00000000000000000000000000000010</td>
<td>+2</td>
</tr>
<tr>
<td>00000000000000000000000000000001</td>
<td>+1</td>
</tr>
<tr>
<td>00000000000000000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>111111111111111111111111111111111</td>
<td>-1</td>
</tr>
<tr>
<td>111111111111111111111111111111110</td>
<td>-2</td>
</tr>
<tr>
<td>........</td>
<td>........</td>
</tr>
<tr>
<td>100000000000000000000000000000001</td>
<td>- 2147483647</td>
</tr>
<tr>
<td>100000000000000000000000000000000</td>
<td>- 2147483648</td>
</tr>
</tbody>
</table>
What happens here?

```java
int i = 32768;
short s = i;
```
5. Lifetime

• “The *lifetime* of a variable is the time during which the variable is bound to a specific memory location.” [p. 219]

• “…the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.” [p. 219]

• Four categories
  – static
  – stack-dynamic
  – explicit heap-dynamic
  – implicit heap-dynamic
Memory organization

STATIC
  code &
  static data

STACK
  local data:
  invocation records

AVAILABLE
  MEMORY

HEAP
  dynamic data
static variables

• Bound to a memory location prior to execution.
• No run-time allocation needs to occur: efficient.
• Persistent: variable persists throughout execution of a program.
#include "stdio.h";  
int counter() {  
    static int k = 0;  
    return ++k;  
}  

int main() {  
    printf("Counter is %d \n",counter());  
    printf("Counter is %d \n",counter());  
    return 0;  
}  

/* OUTPUT IS:  
    Counter is 1  
    Counter is 2  
*/
Notes about example

- static variable k is allocated space in the static segment
- allocation happens once
- k’s lifetime is the that of the entire program
- k’s value persists from call to call
stack dynamic variables

- “storage bindings are created when their declaration statements are elaborated, but whose types are statically bound” [p. 220]
- Allocated on the run-time stack
#include "stdio.h"

int counter() {
    int k = 0;
    return ++k;
}

int main() {
    printf("Counter is %d \n",counter());
    printf("Counter is %d \n",counter());
    return 0;
}

/* OUTPUT IS:
   Counter is 1
   Counter is 1
*/
Notes about example

- stack-dynamic variable k is allocated space in the stack segment
- allocation happens each time function is called
- k’s lifetime is the that of the function invocation
- k’s value does not persist from call to call: it is reinitialized on each function execution
explicit heap-dynamic variables

• anonymous (nameless) variables created at runtime, allocated on the heap, and accessible only via indirection (a pointer)
• Example [p. 221]

```cpp
int *intnode;       // Create a pointer
...
intnode = new int;  // Create the heap-dynamic variable
...
delete intnode;     // Deallocate the heap-dynamic variable
                    // to which intnode points
```
(change from lecture)

• After lecture some students pointed out that ‘new’ is C++, not C. Oddly, Sun’s C compiler compiled the code without complaint.

• I’ve inserted into the slides two versions of the example: a C version (using malloc) and a C++ version (using new).
```c
#include "stdio.h"
#include "stdlib.h"

int counter(int * k) {
    return ++(*k);
}

int main() {
    int * intnode;                       // declare
    intnode = (int *) malloc(sizeof(int)); // allocate
    *intnode = 0;                         // initialize

    printf("Counter is %d \n",counter(intnode));
    printf("Counter is %d \n",counter(intnode));

    return 0;
}

/* OUTPUT IS:
   Counter is 1
   Counter is 1
*/
```cpp
#include <iostream>
using std::cout;
using std::endl;

int counter(int * k) {
    return ++(*k);
}

int main() {
    int * intnode;                          // declare
    intnode = new int(0);                   // allocate & initialize

    cout << "Counter is " << counter(intnode) << endl;
    cout << "Counter is " << counter(intnode) << endl;

    return 0;
}

/* OUTPUT IS:
   Counter is 1
   Counter is 1
*/
```
Notes about C/C++ example

• intnode variable is allocated space on the stack (in main’s invocation record)
• the anonymous variable that intnode points to is allocated space in the heap segment
public class Count {
    private int k;
    public Count() {
        k = 0;
    }
    public int counter() {
        return ++k;
    }
    public static void main(String [] args) {
        Count c = new Count();
        System.out.println("Counter is "+c.counter());
        System.out.println("Counter is "+c.counter());
    }
}

/* OUTPUT IS:
   Counter is 1
   Counter is 2
*/
Notes about Java example

• instance variable k is allocated space in the heap segment
• allocation happens each time object is created
• k’s lifetime is the that of its object
• k’s value persists from call to call of the method
• many different independent k’s can co-exist