More about BOOLEAN issues

- Every boolean test is an implicit comparison against zero (0).
- However, zero is not a simple concept. It represents:
  - the integer zero for all integral types
  - the floating point 0.0 (positive or negative)
  - the null character (‘\0’)
  - the null pointer
- In order to make your intentions clear, explicitly show the comparison with zero for all scalars, floating-point numbers, and characters.

Write an INFINITE LOOP as:

for (;;) ...
while (1) ...

The former is idiomatic among C programmers, and is more visually distinctive.

int i; if (i) is better seen as if (i == 0)
float x; if (!x) is better seen as if (x != 0.0)
char c; if (c) is better seen as (c == '\0')

An exception is made for pointers, since 0 is the only language-level representation for the null pointer.

/* The symbol NULL is not part of the core language - you have to include a special header file to get it defined */

In short, pretend that C has an actual boolean type which is returned by the logical operators and expected by the test constructs, and pretend that the null pointer is a synonym for false.
Unary Operators: ++ --

++a and a++ are both the same as a = a + 1
--a and a-- are both the same as a = a – 1

HOWEVER...
++a → a incremented BEFORE a is used
--a → a decremented BEFORE a is used
a++ → a is incremented AFTER a has been used
a-- → a is decremented AFTER a has been used

In both examples, the final value of a will be 2, BUT...

```c
int main()
{
    int a = 1;
    printf(“ a is %d”, ++a)
    return 0;
}
/* 2 will be printed */
```

```c
int main()
{
    int a = 1;
    printf(“ a is %d”, a++)
    return 0;
}
/* 1 will be printed */
```
Unary Operators ++ and -- (cont)

```
int i = 1, j = 1;
printf("\t%d %d\n", ++i, j++);
printf("\t%d %d\n", i, j);
```

Output:
2 1
2 2

```
i = 0; j = 0;
if ( (i++ == 1) && (j++ == 1))
    printf("what will happen?\n");
printf("\t%d %d\n", i, j);
```

Will i and j get incremented?

```
i=1;  j=1;
printf("\t%d \n",i=j++);
printf("\t%d \n",i=++j);
```

Output:
1
3

2nd printf output:  1 0

The answer is NO! Because the expression in the left of '&&' resolves to false the compiler does NOT execute the expression on the right and so 'j' does not get executed!
**Size of operator**

- `sizeof` will return the number of bytes reserved for a variable or data type.
- Returning the length of a data type (ex.1)
- Length of a variable (ex.2)
- Number of bytes reserved for a structure (ex.3)

**Example 1:**
/* How big is an int? expect an answer of 4. */
main()
{
    printf("%d \n", sizeof(int));
}

**Example 2:**
main()
{
    char String[20];
    printf("%d \n", sizeof String);
    printf("%d \n", sizeof (String));
} // brackets optional but rec

**Example 3:**
/* Will print 8 on most machines. */
main()
{
    struct
    {
        int a;
        int b;
    } TwoInts;

    printf("%d \n", sizeof(TwoInts));
}
Values of variables are stored in memory, at a particular location. A location is identified and referenced with an address, analogous to identifying a house’s location via an address. A pointer is a variable that contains the address of another variable. * is used in the declaration of a pointer type. int *p means variable p is a pointer that points to an integer. & (unary operator) gives the “address of” an object. p = &c means the address of c is assigned to the variable p. * (unary not arithmetic operator) is a dereferencing operator when applied to pointers. When applied to a pointer, it accesses the object the pointer points to. * in front of a pointer variable means “get the value at that address” i.e. “contents of”. int a = *p means get the value at the address designated by p and assign it to. *p = 1 means assign the value of 1 to the memory location designated by the address of p. Every pointer points to a specific data type. Exception = void (a generic pointer); pointer to void holds any type of pointer but can’t be dereferenced (i.e. cannot get the “contents of”).

Ah, yes. POINTERS. At last, we arrive at THE MOST DREADED WORD in the lexicon of the C student. Pointers are indeed so dreaded that Java has completely done away with pointers and wrapped their functionality into the (admittedly safer) concept of references. C++, as a transitional step, has both pointers and references.
Declaring Pointers

- `int* ptr_a;
- `int *ptr_a;
- The first style leads to mistakes
  - `int* ptr_b, ptr_c, ptr_d
    - b is a pointer but c and d are integers
  - `int *ptr_b, *ptr_c, *ptr_d
    - 3 pointers are declared here

Char example
- `char ch = 'c';
- `char *chptr = &ch;
- `char *ptr = chptr;
  - see last example in previous slide
# Pointer example

**Reminders:**
- * in a declaration says “I am a pointer” that points to a certain type of value
- & “address of”
- * In front of a pointer type says “get the value at that address” i.e. “contents of” operator

**EXAMPLE:**

```c
int x=1, y=2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>ADDRESS (in decimal)</th>
<th>MEMORY (assuming 4 bytes per word and each block is a byte)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>0</td>
<td>Is a pointer; holds an addr; 8... 16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>8</td>
<td>1... 0</td>
</tr>
<tr>
<td>y</td>
<td>12</td>
<td>2... 1</td>
</tr>
<tr>
<td>z</td>
<td>16</td>
<td>z[0]</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>z[1]</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>z[2]</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>etc</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

* not going to worry about "size" right now