- Introductory remarks.
  - Knowing C/C++ (as opposed to other high-level languages like Java) should put one in an advantageous position when learning assembly language because assembly language is a *low-level* language and C/C++ has features that put it at a *lower level* among the high-level languages. Prime examples of such features:
    - *array* and *pointer*, and the associated *subscript operator* (**[ ]**), *address-of operator* (**&**) and *dereference operator* (*****).
    - bitwise operators (**~**, **&**, **|**, **^**, **<<** and **>>**); these, as opposed to **!**, **&&** and **| |**, are akin to MIPS instructions.
  - To enjoy the advantage, however, one must have mastered the said features when learning C/C++.
    - Observations on why some past students failed to enjoy (or fully enjoy) the advantage:
      - Not quite mastering the features involved (in prior classes).
        - » Prior knowledge of *bitwise operations* not expected → make up via ***005a NotesOnBitwiseOperations***.
      - 2 gray areas (excluding *bitwise operations* for reason indicated above) students tend to have difficulty:
        - » For a *non-pointer* variable appearing in an expression → *lvalue* versus *rvalue*.
          - Examples (`x` and `a[5]` are of type `int`): `x = x + 2` and `a[5] = a[5] + 2`
        - » For a *pointer* appearing (individually or in tandem with *****) in an expression → *indirection* (or lack thereof) coupled with *lvalue* versus *rvalue*, potentially also involving *pointer arithmetic*.
          - Understanding pointer semantics, for *e.g.* (`xPtr` is a pointer of type `int*`):

            ```
            xPtr = xPtr + 2
            *xPtr = *xPtr + 2
            *xPtr = *(xPtr + 2)
            ```

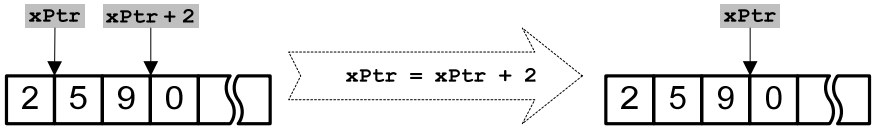
          - Using pointers (*pointer hopping*) to traverse/process arrays, or rewriting in equivalent pointer form.
            - Mapping *pointer-based* (as opposed to *index-based*) C++ code into MIPS AL is simpler since a *pointer* maps directly to a register (but it is not the case with an *indexed variable* like `a[i]`).
          - Using pointers to effect *passing by reference*, or rewriting in equivalent pass by address form.
            - *Pass by reference* works as a convenient (but also confusing?) abstraction of C's *pass by address*.
            - Being lower level, the less abstract *pass by address* fits the bill for MIPS AL; reference type simply doesn't exist in MIPS AL.

- Quick review of select gray-area items.
  - *lvalue* versus *rvalue* for *non-pointer* variable (for simplicity, variable `x` of type `int` referenced below).
    - Ask and decide which 1 of 2 situations applies (makes sense):
      - Need for `x` to be *memory location*.
      - Need for `x` to be *<u>value contained in</u> memory location*.

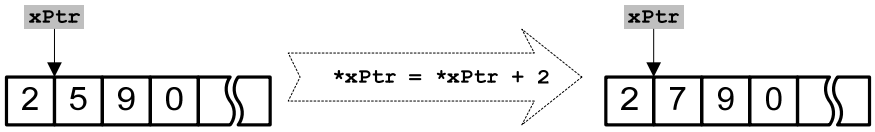
    Continuing with `x = x + 2` and `a[5] = a[5] + 2` examples above:
      - LHS `x` and `a[5]` need to be *memory location* named `x` and `a[5]`, respectively.
      - RHS `x` and `a[5]` need to be *<u>value contained in</u> memory location* named `x` and `a[5]`, respectively.
  - *lvalue* versus *rvalue* for *pointer* (for simplicity, pointer `xPtr` of type `int*` referenced below).
    - Ask and decide which 1 of 2 situations applies (makes sense):
      - For `xPtr` appearing <u>not in tandem with *****</u>.
        - » Need for `xPtr` to be *memory location*.
        - » Need for `xPtr` to be *<u>value contained in</u> memory location*.

- For `xPtr` appearing <u>in tandem with *****</u>.
  - » Need for `*xPtr` to be "*memory location* pointed at by `xPtr`".
  - » Need for `*xPtr` to be <u>*value contained in*</u> "*memory location* pointed at by `xPtr`".

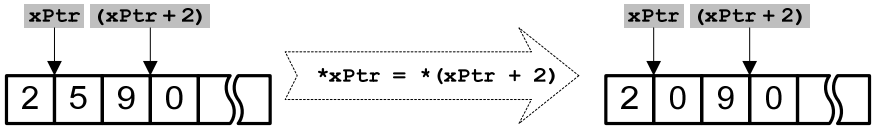
Continuing with `xPtr = xPtr + 2` example above:

- LHS `xPtr` needs to be *memory location* named `xPtr`.
- RHS `xPtr` needs to be <u>*value contained in*</u> *memory location* named `xPtr`.
- 

Continuing with `*xPtr = *xPtr + 2` example above:

- LHS `*xPtr` needs to be "*memory location* pointed at by `xPtr`".
- RHS `*xPtr` needs to be <u>*value contained in*</u> "*memory location* pointed at by `xPtr`".
- 

Continuing with `*xPtr = *(xPtr + 2)` example above:

- LHS `*xPtr` needs to be "*memory location* pointed at by `xPtr`".
- RHS `*(xPtr + 2)` needs to be <u>*value contained in*</u> "*memory location* pointed at by `(xPtr + 2)`".
- 

- Using pointers to traverse/process array, an example:
  - Via *index* and **[ ]** operator (*index-based*, for comparison):

```
for (int i = 0; i < 300; ++i)
{
   if (i < 100)
      a2[i] = (a1[i] + a1[i + 1]) / 2;
   else if (i < 200)
      a2[i] = (a1[i - 1] + a1[i] + a1[i + 1]) / 3;
   else
      a2[i] = (a1[i - 1] + a1[i]) / 2;
}
```

  - Via *pointers* and ***** operator (*pointer-based*):

```
int* a1HopPtr  = &a1[0];       // SAME AS: int* a1HopPtr  = a1;
int* a1EndPtr1 = &a1[100];     // SAME AS: int* a1EndPtr1 = a1 + 100;
int* a1EndPtr2 = &a1[200];     // SAME AS: int* a1EndPtr2 = a1 + 200;
int* a1EndPtr3 = &a1[300];     // SAME AS: int* a1EndPtr3 = a1 + 300;
int* a2HopPtr  = &a2[0];       // SAME AS: int* a2HopPtr  = a2;
while (a1HopPtr < a1EndPtr3)
{
   if (a1HopPtr < a1EndPtr1)
      *a2HopPtr = (*a1HopPtr + *(a1HopPtr + 1)) / 2;
   else if (a1HopPtr < a1EndPtr2)
      *a2HopPtr = ((*(a1HopPtr - 1) + *a1HopPtr + *(a1HopPtr + 1)) / 3;
   else
      *a2HopPtr = (*(a1HopPtr - 1) + *a1HopPtr) / 2;
   ++a1HopPtr;
   ++a2HopPtr;
}
```

- Using pointers to effect passing by reference, an example:
  - Via *pass by reference* (for comparison):
    - How `Swap` is called:
    - How `Swap` is implemented:

```
Swap(i1, i2);
```

```
void Swap(int& i1Ref, int& i2Ref)
{
   int temp = i1Ref;
   i1Ref = i2Ref;
   i2Ref = temp;
}
```

  - Via *pass by address*:
    - How `Swap` is called:
    - How `Swap` is implemented:

```
Swap(&i1, &i2);
```

```
void Swap(int* i1Ptr, int* i2Ptr)
{
   int temp = *i1Ptr;
   *i1Ptr = *i2Ptr;
   *i2Ptr = temp;
}
```

- How concepts relate to MIPS AL:
  - Operation (*e.g.*: assignment, input) involving *lvalue* → store instruction needed.

    Operation (*e.g.*: arithmetic/logical, output) involving *rvalue* → load instruction needed.

    Continuing with `x = x + 2` example above:

```
la $t0, x               # $t0 has address of x
lw $t1, 0($t0)          # $t1 has "current value of x" (RHS x as rvalue)
addi $t1, $t1, 2        # $t1 now has "current value of x" + 2
sw $t1, 0($t0)          # x gets its prior value + 2 (LHS x as lvalue)
```

    Continuing with `a[5] = a[5] + 2` example above:

```
la $t0, a               # $t0 has address of a
lw $t1, 20($t0)         # $t1 has "current value of a[5]" (RHS a[5] as rvalue)
addi $t1, $t1, 2        # $t1 now has "current value of a[5]" + 2
sw $t1, 20($t0)         # a[5] gets its prior value + 2 (LHS a[5] as lvalue)
```

    Examples involving pointers:

    Notes on C++ variables involved and MIPS register usage:

    `$a0` as `a0Ptr` (a pointer containing address of a "*4-byte*-`int`")
    `$a1` as `a1Ptr` (a pointer containing base address of an array of "*4-byte*-`int`"s)
    `$a2` as `a2Ptr` (a pointer containing base address of another array of "*4-byte*-`int`"s)
    `$v0` as `v0` (a "*4-byte*-`int`")
    `$a3`, `$t0`, `$v1` (temporary holders for "*4-byte*-`int`" or address)

`a0Ptr = a1Ptr + 2`

```
addi $a0, $a1, 8        # no pointer (only "simple") arithmetic in assembly
```

`*a0Ptr = v0 + 2`

```
addi $t0, $v0, 2        # $t0 has v0 + 2
sw $t0, 0($a0)          # *a0Ptr as lvalue
```

`*a0Ptr = *a0Ptr + 2`

```
lw $v1, 0($a0)          # $v1 gets *a0Ptr (RHS *a0Ptr as rvalue)
addi $v1, $v1, 2        # $v1 now has *a0Ptr + 2
sw $v1, 0($a0)          # *a0Ptr gets *a0Ptr + 2 (LHS *a0Ptr as lvalue)
```

`*(a1Ptr - 1) = *(a1Ptr + 2)`

```
lw $t0, 8($a1)          # $t0 has *(a1Ptr + 2) as rvalue
sw $t0, -4($a1)         # *(a1Ptr - 1) as lvalue gets *(a1Ptr + 2)
```

`++(*a0Ptr)` (has same effect as **`*a0Ptr = *a0Ptr + 1`**, so translate it as such)

```
lw $v1, 0($a0)          # $v1 has *a0Ptr as rvalue
addi $v1, $v1, 1        # $v1 now has "current value of *a0Ptr" + 1
sw $v1, 0($a0)          # *a0Ptr as lvalue gets "current value of *a0Ptr" + 1
```

`*(a2Ptr + *a0Ptr) = *(a1Ptr + v0)`

```
sll $t0, $v0, 2         # $t0 has 4 * v0
add $t0, $t0, $a1       # $t0 now has a1Ptr + v0 (pointer arithmetically)
lw $v1, 0($t0)          # $v1 has *(a1Ptr + v0) as rvalue
lw $a3, 0($a0)          # $a3 has *a0Ptr as rvalue
sll $a3, $a3, 2         # $a3 now has 4 * (*a0Ptr)
add $a3, $a3, $a2       # $a3 now has a2Ptr + *a0Ptr (pointer arithmetically)
sw $v1, 0($a3)          # *(a2Ptr + *a0Ptr) as lvalue gets *(a1Ptr + v0)
```

- Storage.

  - In C++, for portability, only *main memory* (RAM) is available for use as storage and we may use:
    - Named storage with *static* lifetime allocated at compile time (*e.g.*: *global* or *static local* variable).
    - Named storage with *automatic* lifetime allocated at compile time (*e.g.*: automatic *local* variable).
    - Unnamed storage with "`new`-*till*-`delete`" lifetime allocated at runtime (*e.g.*: dynamic array).
    - Each storage may be *scalar* (*e.g.*: `char` or `int` variable) or *composite* (*e.g.*: array).
    - Each storage has *type* (`char`, `int`, *etc.*) and *mutability* (variable or constant) associated with it.

  In MIPS AL, both *register*s and *main memory* are available for use as storage:

  - Neither *type* nor *mutability* is associated with any storage (*i.e.*, all storage is *untyped*).
    - Can use a register or any part of (allocated) *main memory* to store an integer, a character, an address. *etc.*
      - » *Implicit type* of storage is interpreted according to the instruction that uses it.
      - » Assembler does not keep track of any type information for data (referenced by variables/addresses).
        - → A key consequence of this is that there is *no pointer arithmetic* (*i.e.*, only "simple" arithmetic) in AL (since type-dependent "per-item storage requirement" is needed to do pointer arithmetic).
    - Programmer's responsibility to maintain any *intended immutability* of storage.
      - » Except *Register 0* (`$0` or `$zero`) – it is hardwired to always contain 0 (even after being written to).
  - No operand of any instruction can be *main memory* storage.
    - May only be *register* or *immediate value* (constant embedded in instruction).
    - MIPS ISA → *load-store* architecture → only *load* and *store* instructions access *main memory*.
      - » Operands of *load* and *store* instructions still have to be *register* or *immediate value*.
  - Want to use register for *scalar* storage (including *pointer*) as far as possible (for speed).
    - Unless storage needs to be *passed by address* – a register has no address (and it's *visible to all* anyways).
    - Due to limited number of registers:
      - » May have to use *main memory* for some (less frequently accessed) scalar storage.
      - » Can also *spill register* – save register's content to *main memory* (so register is free to be used for another purpose) and restore register's content from *main memory* later when needed.
  - Want to use *main memory* for an *array*.
    - Array is typically big and its elements are sequentially (contiguously) addressable storage locations.
    - Want to *traverse* array (access each element) via a *computable target address* (can't do so with registers).
    - Put array with *static lifetime* (*e.g.*, global array) in *data segment* and put array with *automatic lifetime* (*e.g.*, non-static local array) in *stack segment*.
  - Have to be mindful of *memory alignment* when using *main memory* storage.