1. PC’s are byte-addressable: each byte has an address (1 byte == 8 bits)

```c
float fv;     // 4 bytes
int iv;      // 4 bytes
char cv;    // 1 byte
```

<table>
<thead>
<tr>
<th>variables</th>
<th>addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>cv</td>
<td>0012FF01</td>
</tr>
<tr>
<td>iv</td>
<td>0012FF02</td>
</tr>
<tr>
<td></td>
<td>0012FF03</td>
</tr>
<tr>
<td></td>
<td>0012FF04</td>
</tr>
<tr>
<td></td>
<td>0012FF05</td>
</tr>
<tr>
<td>fv</td>
<td>0012FF06</td>
</tr>
<tr>
<td></td>
<td>0012FF07</td>
</tr>
<tr>
<td></td>
<td>0012FF08</td>
</tr>
<tr>
<td></td>
<td>0012FF09</td>
</tr>
</tbody>
</table>

Variables are used in high-level languages.
Numeric addresses are used by the machine.

2. Addresses are displayed as a hexadecimal number.
Address is made up of a segment: 0012
and an offset: FF01

**Segment:** identifies a block of memory
**Offset:** represents memory position relative to beginning of segment

3. **Address operator** (&): unary operator that gives the address of its operand
Address of a variable is address of **first byte** of the variable

```c
&cv is 0012FF01
&iv is 0012FF02
&fv is 0012FF06
```

4. **pointer:** address of a memory location

value of a pointer is the address of another variable or another area of memory

**Note:** Pointer is a compound type. You can have a pointer constant or a pointer variable.
// AddOp: Address Operator

#include <iostream>  // cout

using namespace std;

int main()
{
    short sv = 50;
    int iv = 100;
    float fv = 2.25F;  // Note suffix
    double dv = 2.5;

    cout << "Contents Address Size of Size of
    of Variable: of Variable: Variable: Address:

    " << sv << "\t\t" << &sv << "\t" << sizeof sv << "\t\t"
    << sizeof &sv << "\n\n";

    cout << "int:\n" << iv << "\t\t" << &iv << "\t" << sizeof iv << "\t\t"
    << sizeof &iv << "\n\n";

    cout << "float:\n" << fv << "\t\t" << &fv << "\t" << sizeof fv << "\t\t"
    << sizeof &fv << "\n\n";

    cout << "double:\n" << dv << "\t\t" << &dv << "\t" << sizeof dv << "\t\t"
    << sizeof &dv << "\n";

    return 0;
}

Output:

<table>
<thead>
<tr>
<th>Contents of Variable</th>
<th>Address</th>
<th>Size of Variable</th>
<th>Size of Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>short:</td>
<td>50</td>
<td>0012FF74</td>
<td>2</td>
</tr>
<tr>
<td>int:</td>
<td>100</td>
<td>0012FF7C</td>
<td>4</td>
</tr>
<tr>
<td>float:</td>
<td>2.25</td>
<td>0012FF78</td>
<td>4</td>
</tr>
<tr>
<td>double:</td>
<td>2.5</td>
<td>0012FF6C</td>
<td>8</td>
</tr>
</tbody>
</table>
1. **Pointer variable**: variable that stores the address of a value

2. Declaring pointer variables:

   ```
   10 char* pc;       // pc is a pointer to char
   20 int* pi;        // pi is a pointer to int
   30 float* pf1, * pf2; // pf1 and pf2 are pointers to float
   40 short* ps1, ps2; // ps1 is pointer to short; ps2 is short
   ```

   Data type indicates the pointed-to-type:

   ```
   int* (pointer-to-int) is a compound type; built from int
   ```

3. ```
   50 int num = 3, x = 4;
   60 int* pnum;       // pnum is pointer to int
   70 pnum = &num;     // assigns address of num to pnum
   ```

   "pnum points to num"
a. &num and &x are pointer constants.
b. pnum is a variable. pnum could point to another int.

\[ \text{pnum} = \&x; \quad \text{// now pnum points to x} \]

Note that address and pointer are synonyms.

4. \begin{verbatim}
80   int num = 3, x = 4;
90   int* pnum;  // pointer to int
100  pnum = &num;  // assign address of num to pnum
110  x = *pnum;   // assign pointed-to-value to x
120  // Indirection operator or dereferencing operator
130  // unary operator that gives pointed-to-value
140  // Indirect way of: x = num;
150  *pnum = 5;   // assign 5 to num; Dereference the pointer
160  x = *pnum - 1; // assign num - 1 to x
170  ++*pnum;  // same as: ++(*pnum);
180  // ++ and *(indirection) have same precedence;
190  // R-L asso.
\end{verbatim}

Always a pointer to a type; no generic pointer. Why?

To access the pointed-to-object:
 a. to access the correct amount of storage
 b. to use the correct interpretation of the bits

\begin{verbatim}
int num
num
x &num  \quad \text{Note that &num and &x are constants}
*pnum &x
\end{verbatim}
5. int num;
   int* pnum;

   Primary Quantity  Compound Quantity
   num      int   &num      int*
   pnum      int*   *pnum      int

6. int* pi;   // local pointer variable
               
               *pi = 25;   // pi has not been set to a proper address;
               // pi is not pointing to anything

   Probably overwrite own program.

   Protected Environment:    program terminates
   Unprotected Env.:         overwrite anything (OS)

7. Naming conventions:     200  int number;
   210  int* pnumber;
   220  int* numberptr;
   230  double rate;
   240  double* prate;

   Style
   int  *iptr;   // C      *iptr is an int   is emphasized
   int* iptr;   // C++     int* is a type; pointer-to-int; choice of Stroustrup
   int * iptr;   // Politically Correct

8. Pointers are not integers.

   Integers can be added, subtracted, multiplied, and divided.
   Pointers describe a location;     cannot be multiplied or divided

   int* iptr;
   iptr = 0XBF1000;     // type mismatch: int to int*

   iptr = static_cast<int*>(0XBF1000);  // use cast
// Swap: Swapping the contents of actual arguments
#include <iostream> // cout
using namespace std;

void swap_by_value(int, int); // Prototypes
void swap_by_reference(int&, int&);
void swap_with_pointers(int*, int*);

int main()
{
    int a = 100, b = 22;
    cout << "Before first swap:\n"
    << " a = " << a << " \tb = " << b << "\n\n";
    swap_by_value(a, b); // Actual argument can be a constant, 
    // variable, or expression
    cout << "After swap by value:\n"
    << " a = " << a << " \tb = " << b << "\n\n";
    swap_by_reference(a, b); // Actual argument should be a variable
    cout << "After swap by reference:\n"
    << " a = " << a << " \tb = " << b << "\n\n";
    swap_with_pointers(&a, &b);
    cout << "After swap with pointers:\n"
    << " a = " << a << " \tb = " << b << "\n";
    return 0;
}

void swap_by_value(int x, int y) // Definitions
{
    int temp = x;
    x = y;
    y = temp;
}

void swap_by_reference(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

void swap_with_pointers(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
Output:  
Before first swap:  
a = 100 b = 22  

After swap by value:  
a = 100 b = 22  

After swap by reference:  
a = 22 b = 100  

After swap with pointers:  
a = 100 b = 22  

Stack at beginning of swap_by_value(a, b):  
\[
\begin{array}{c|c|c|c}
\text{temp} & \text{copy of a} & x & \text{swap_by_value()} \\
\hline
100 & 100 & x & \\
22 & 22 & y & \\
\end{array}
\]

main()  
\[
\begin{array}{c|c|c}
a & 100 & b \\
\hline
\end{array}
\]

Stack at beginning of swap_by_reference(a, b):  
\[
\begin{array}{c|c|c|c}
\text{temp} & \text{swap_by_reference()} & \text{copy of a} & x \\
\hline
b & 22 & 100 & \\
a & 100 & y & \\
\end{array}
\]

Stack at beginning of swap_with_pointers(&a, &b):  
\[
\begin{array}{c|c|c|c|c}
\text{temp} & \text{swap_with_pointers()} & \text{copy of a} & x & \text{copy of b} \\
\hline
b & 100 & 22 & y \\
a & 22 & x & \\
\end{array}
\]
1. Usually used in initialization and update expressions of for loops.

2.  
   ```
   int i, sum, a, b;
   
   sum = 0;
   a = 2;
   b = 3;
   for (i = 1; sum < 1000; i++)
   {
      sum += i * (a + b);
      a += 2;
      b += 3;
   }
   ```

   for (sum = 0, a = 2, b = 3, i = 1; sum < 1000; sum += i * (a + b), a += 2, b += 3, i++)
   continue; // NULL statement or continue;

   Comma operator has the lowest precedence and left-to-right associativity.

   for (sum = 0, a = 2, b = 3, i = 1; sum < 1000; sum += i * (a + b), a += 2, b += 3, i++)
   continue; // NULL statement or continue;

   continue statement transfers control to the update expression in a for and to the condition in while and do...while loops.

3. Value of whole comma expression is value of rightmost member.
   a. ```
      a = (y = 8, x = 12);
      ``` equivalent to
      ```
      y = 8; x = 12; a = 12; // Value of comma exp is 12
      ```
   b. ```
      a = (b = 12, b + 8);
      ``` equivalent to
      ```
      b = 12; a = b + 8; // Value of comma exp is 20
      ```

4. Comma is also used as a separator.
   ```
   int a, b, c; or fun(a, b, c, 2);
   ```

   Separators, not comma operators, thus, not a sequence point (, ; & & || end of exp).