Programming 2: Memory

Sam Meehan
CERN
30 January 2020
Warnings/Pointers

● Protip: Communication is key (not only in physics)
  ○ The more responsive you are, the more successful you will be
  ○ e.g. email list

● On the lesson
  ○ Stop me with questions throughout
    ■ Remember, I can’t see your faces
  ○ Best to follow along on the posted slides
    ■ Screenshots may be small …
  ○ There are pictorial diagrams
    ■ These are useful to begin with (SUPER USEFUL) but will not persist

2

csu-fresno-summer-2020@cern.ch
Survey

Have you ever explored/learned programming on your own time prior to this course?

12 responses

- Yes: 91.7%
- No: 8.3%
Survey

How comfortable are you with the C++ programming language?

12 responses

1: 9 (75%)
2: 3 (25%)
3: 0 (0%)
Survey

Which is more precise, an integer or a floating point number?

12 responses

Correct = Integer
Survey

[Refer to the above image of two code snippets] Which of the two (or perhaps both) code sections will correctly perform a swap of the variables x and y? If one of them will not work, describe why it will not work.

12 responses

Correct = Option 2
Introduction

- **Computer memory**:  
  - How to think about it  
  - How to use it in a basic way  
  - How to use it in a more advanced way  
    - “operations”

- **How to read programming documentation**:  
  - The important bits for now  
  - Ultimate goal: Reading/navigating ROOT documentation

- **What you should be educating yourself about**:  
  - Glazing over things here that I consider “secondary”

- **BONUS/Extra**: What is actually being done in the machine?
Memory

● Everything in a computer must be formatted in binary in some way: 0,1 = “bits”
  ○ Relies on ASCII conventions about how this is done
  ○ ASCII = dictionary to translate keyboard letters, symbols, etc. into machine language (0,1)
  ○ 1 Gb = 1 Giga-byte (Giga = 10^9, byte = 8 bits) → 8*10^9 spaces for either 0 or 1

● Example: integers
  ○ What is 14 in binary? : 14 < 16(=2^4) → 8+4+2 → 1*2^3+1*2^2+1*2^1+0*2^0 →
  ○ Higher number = more memory
    ■ “max integer” of 2147483647 = 2^31-1
    ■ (31 bits for size) + (1 bit for ±)

● Spelling: 26 letters + space → 5 bits
  ○ Can write sentences if we make convention of chunking every 5 bits

Convert letters into blocks of bits “byte”

I like to ski in the winter
01001000000110101001011000010100000101011000000000101000110001001000000100101111000001010101000001100010111110101001011001
Memory

- Everything in a computer must be formatted in binary in some way: 0,1 = “bits”
  - Relies on ASCII conventions about how this is done
  - ASCII = dictionary to translate keyboard letters, symbols, etc. into machine language (0,1)
  - 1 Gb = 1 Giga-byte (Giga = 10⁹, byte = 8 bits) → 8*10⁹ spaces for either 0 or 1

- Example: integers
  - What is 14 in binary? : 14 < 16 (=2⁴) → 8+4+2 → 1*2³+1*2²+1*2¹+0*2⁰ → 1 1 1 0
  - Higher number = more memory
    - "max integer" of 2147483647 = 2³¹-1
    - (31 bits for size) + (1 bit for ±)

- Spelling: 26 letters + space → 5 bits
  - Can write sentences if we make convention of chunking every 5 bits
  
Convert letters into blocks of bits “byte”

Small corruption (additional “1” on front) produces gibberish

I like to ski in the winter

tpftvbpjx jfdpdwpjtpb dwzry
Everything in a computer must be formatted in binary in some way: 0,1 = “bits”
  - Relies on ASCII conventions about how this is done
  - ASCII = dictionary to translate keyboard letters, symbols, etc. into machine language (0,1)
  - 1 Gb = 1 Giga-byte (Giga = $10^9$, byte = 8 bits) → $8 \times 10^9$ spaces for either 0 or 1

Example: integers
  - What is 14 in binary? : $14 < 16 (=2^4)$ → $8+4+2$ → $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ → 1110
  - Higher number = more memory
    - “max integer” of 2147483647 = $2^{31} - 1$
    - (31 bits for size) + (1 bit for ±)

~Concept of “the stack”

Generic picture to develop an intuition for: “chunks of memory”
Two interesting concepts concerning memory result from this picture:

- **“Pointers”**: the thing stored in memory tells you the location of another bit of memory.
- **“Memory Leak”**: if your program somehow just keeps assigning memory to new “things” in an uncontrolled way.

![Memory Diagram]

- `thing5 = location of thing2 = 32`
Memory - Funny Concepts

Two interesting concepts concerning memory result from this picture:

- "Pointers": the thing stored in memory tells you the location of another bit of memory.
- "Memory Leak": if your program somehow just keeps assigning memory to new "things" in an uncontrolled way.

Working to develop this picture in your head is important. Until you can do that, don’t be afraid to get a pen and paper and draw memory.
Integers

- One type of memory structure
  - Allocation of space according to the type of structure
- Key parts to “making” one of these
  - **type**: in this case “int”
  - **name**: example “myXvariable”
  - **initialization**: what the initial value is that is stored in the place in memory

```
int myXvariable = 6;
int myYvariable = 4;
int sum = 0;
sum = myXvariable + myYvariable;
```

Created and initialized in header

```
#include <iostream>
int main()
{
  int myXvariable = 6;
  int myYvariable = 4;
  cout << "The sum is: " << sum << endl;
  cout << "Hello CERN" << endl;
  return 0;
}
```

Created and initialized in source code
Integers ... but there's more

- Multiple (but limited) set of **predefined memory structures in c++**
  - Everything else is built from these!

<table>
<thead>
<tr>
<th>Group</th>
<th>Type names*</th>
<th>Notes on size / precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character types</td>
<td>char</td>
<td>Exactly one byte in size. At least 8 bits.</td>
</tr>
<tr>
<td></td>
<td>char16_t</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td></td>
<td>char32_t</td>
<td>Not smaller than char16_t. At least 32 bits.</td>
</tr>
<tr>
<td></td>
<td>wchar_t</td>
<td>Can represent the largest supported character set.</td>
</tr>
<tr>
<td>Integer types (signed)</td>
<td>signed char</td>
<td>Same size as char. At least 8 bits.</td>
</tr>
<tr>
<td></td>
<td>signed short int</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td></td>
<td>signed int</td>
<td>Not smaller than short. At least 16 bits.</td>
</tr>
<tr>
<td></td>
<td>signed long int</td>
<td>Not smaller than int. At least 32 bits.</td>
</tr>
<tr>
<td></td>
<td>signed long long int</td>
<td>Not smaller than long. At least 64 bits.</td>
</tr>
<tr>
<td>Integer types (unsigned)</td>
<td>unsigned char</td>
<td>(same size as their signed counterparts)</td>
</tr>
<tr>
<td></td>
<td>unsigned short int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td></td>
</tr>
<tr>
<td>Floating-point types</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>Precision not less than float</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>Precision not less than double</td>
</tr>
<tr>
<td>Boolean type</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>Void type</td>
<td>void</td>
<td>no storage</td>
</tr>
<tr>
<td>Null pointer</td>
<td>decltype(nullptr)</td>
<td></td>
</tr>
</tbody>
</table>
Casting - “data type manipulation”

- Cast (Merriam-Webster) : to form by this process
  - Casting changes the interpreted form of the memory that is being stored
- Manipulation of data types can be useful (but dangerous)
  - Necessary if a function takes a float as an argument but you have data as a string
Type Casting of Fundamental Types

- Conversion between fundamental numerical types is “intuitive”
  - Implicit (sloppy): the compiler will take care of it
  - Explicit (careful): I will specify exactly what this variable ends up as
    - [1]“c-like notation” = (new_type) var_to_cast
    - [2]“Functional notation” : new_type(var_to_cast)

(Smaller $\rightarrow$ Larger): “Promotion”

Exact value retained

(Larger $\rightarrow$ Smaller)

Case specific result
Float $\rightarrow$ Int results in truncation
Type Casting of Fundamental Types

- Conversion between fundamental numerical types is “intuitive”
  - Implicit (sloppy): the compiler will take care of it
  - Explicit (careful): I will specify **exactly** what this variable ends up as

These two methods do **the same thing**
Operations

- Predefined set of operations are recognized during compilation
  - Plus, minus, multiply, divide ...
- What they do depends on context but usually this is intuitive
  - $+$: “INT + INT” is recognized at compilation
  - $=$: “assignment” operator is not “equals”
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[\begin{array}{c}
10 \\
+ 11 \\
\hline
111 \\
\end{array}\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[\begin{array}{c}
10 \\
+ \quad 11 \\
\hline
\phantom{1}1
\end{array}\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
0 \\
10 \\
+ 11 \\
\hline
1
\end{array}
\]
Low Level Addition

• But how is the computer doing addition at the bit level?
  ○ The same way as we do addition on paper (in base 10) but in base 2

• Two step process
  ○ [1] Add the same bit
  ○ [2] Carry if there is a left over
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ \\
11 \\
\hline
21 \\
\end{array}
\quad \quad \quad \quad \quad
\begin{array}{c}
19 \\
+ \\
13 \\
\hline
32 \\
\end{array}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ \hspace{1em} 11 \\
\hline
21 \\
\end{array} \\
\begin{array}{c}
19 \\
+ \hspace{1em} 13 \\
\hline
32 \\
\end{array}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{align*}
\text{10} & \quad + \quad \text{11} \\
\hline
\text{21} & \quad \text{1010} \\
\hline
\text{21} & \quad \text{1011}
\end{align*}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ \ 11 \\
\hline
21 \\
\end{array}
\quad \quad \quad
\begin{array}{c}
1010 \\
+ \ 1011 \\
\hline
10111 \\
\end{array}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ 11 \\
\hline
21 \\
\end{array}
\quad
\begin{array}{c}
1010 \\
+ 1011 \\
\hline
1011 \\
\end{array}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{align*}
10 & \quad + \quad 11 \\
\hline
21 & \\
\end{align*}
\]

\[
\begin{align*}
1010 & \quad + \quad 1011 \\
\hline
21 & \\
\end{align*}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

In a computer we can’t count to 2!!!!! We can only count up to 1, from 0. Everything else is *interpreting the string of {0,1}.
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ 11 \\
\hline
21
\end{array}
\quad
\begin{array}{c}
1010 \\
+ 1011 \\
\hline
1011
\end{array}
\]
**Low Level Addition**

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ 11 \\
\hline
21 \\
\end{array}
\]

\[
\begin{array}{c}
1010 \\
+ 1011 \\
\hline
2101 \\
\end{array}
\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2

- Two step process
  - [1] Add the same bit
  - [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ 11 \\
\hline
21 \\
\end{array}
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{array}
\begin{array}{c}
1 \\
01010 \\
+ 01011 \\
\hline
0101 \\
\end{array}
But how is the computer doing addition at the bit level?
  ○ The same way as we do addition on paper (in base 10) but in base 2

Two step process
  ○ [1] Add the same bit
  ○ [2] Carry if there is a left over

\[
\begin{array}{c}
10 \\
+ 11 \\
\hline
21
\end{array}
\]

\[
\begin{array}{c}
101010 \\
+ 01011 \\
\hline
101011
\end{array}
\]

\[16 + 0 + 4 + 0 + 1 = 21\]
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing 0+1 at bit level??!?!?!?!
  - The computer is nothing more than an electric (digital!) circuit
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing 0+1 at bit level??!?!?!?
  - The computer is nothing more than an electric (digital!) circuit
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing $0+1$ at bit level??!?!?!?!
  - The computer is nothing more than an electric (digital!) circuit

```
  1010
  1011
```

```
  1010
  1011
  AND
  Carry Bit
```
**Low Level Addition**

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing 0+1 at bit level??!?!?!?!
  - The computer is nothing more than an electric (digital!) circuit
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing 0+1 at bit level??!?!?!?!
  - The computer is nothing more than an electric (digital!) circuit
Low Level Addition

- But how is the computer doing addition at the bit level?
  - The same way as we do addition on paper (in base 10) but in base 2
- But how is the computer doing 0+1 at bit level??!?!?!?!?
  - The computer is nothing more than an electric (digital!) circuit

You will likely never have to think this low level in HEP unless you want to go into the design of Microelectronics and ASICs (Application Specific Integrated Circuits)
So Many Operators

- Many different types of operators are useful in different places
  - Be aware that they exist - good overview on cplusplus.com

<table>
<thead>
<tr>
<th>Level</th>
<th>Precedence group</th>
<th>Operator</th>
<th>Description</th>
<th>Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scope</td>
<td>: :</td>
<td>scope qualifier</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>Postfix (unary)</td>
<td>++ --</td>
<td>postfix increment / decrement</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>()</td>
<td>Functional forms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[]</td>
<td>subscript</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>. -&gt;</td>
<td>member access</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Prefix (unary)</td>
<td>++ --</td>
<td>prefix increment / decrement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1</td>
<td>bitwise NOT / logical NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ -</td>
<td>unary prefix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; *</td>
<td>reference / dereference</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>new delete</td>
<td>allocation / deallocation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sizeof</td>
<td>parameter pack</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(type)</td>
<td>C-style type-casting</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pointer-to-member</td>
<td>, -&gt; *</td>
<td>access pointer</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>5</td>
<td>Arithmetic: scaling</td>
<td>/ %</td>
<td>multiply, divide, modulo</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>6</td>
<td>Arithmetic: addition</td>
<td>+</td>
<td>addition, subtraction</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>7</td>
<td>Bitwise shift</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>shift left, shift right</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>8</td>
<td>Relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>comparison operators</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>9</td>
<td>Equality</td>
<td>== !=</td>
<td>equality / inequality</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>10</td>
<td>And</td>
<td>&amp;</td>
<td>bitwise AND</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>11</td>
<td>Exclusive or</td>
<td>^</td>
<td>bitwise XOR</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>12</td>
<td>Inclusive or</td>
<td></td>
<td></td>
<td>Left-to-right</td>
</tr>
<tr>
<td>13</td>
<td>Conjunction</td>
<td>&amp; &amp;</td>
<td>logical AND</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>14</td>
<td>Disjunction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Assignment-level expressions</td>
<td>= /= &amp;= ^=</td>
<td>assignment / compound assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>16</td>
<td>Sequencing</td>
<td>,</td>
<td>comma separator</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>

This is the origin of the name “c++”:
“c++ ←→ c + stuff”
Exercise: Memory Chart

Draw a memory flow on a piece of paper in front of you
Exercise: Memory Chart

Draw a memory flow on a piece of paper in front of you.

```
int main()
{
    int a, b;
    cin >> a; cin >> b;
    cout << "a: " << a;
    cout << " b: " << b;'n
    cout << "c: " << a + b;
    cout << "d: " << a - b;
    return 0;
}
```
Exercise: Memory Chart

Draw a memory flow on a piece of paper in front of you.

(L7)  
```
<table>
<thead>
<tr>
<th>a(int)</th>
<th>b(int)</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
```

(L8)  
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
```

(L9)  
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
```

(L10)  
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>??</td>
</tr>
</tbody>
</table>
```

(L11)  
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>??</td>
</tr>
</tbody>
</table>
```

(L12)  
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>??</td>
</tr>
</tbody>
</table>
```
Survey Revisit

[Refer to the above image of two code snippets] Which of the two (or perhaps both) code sections will correctly perform a swap of the variables x and y? If one of them will not work, describe why it will not work.

12 responses

Correct = Option 2
Collections of Data

- Rarely do you want to store just one of some value
  - Example: ATLAS pixel detector has ~90,000,000 channels - 90,000,000 separate floats for the energy in each for every events ➔ that would be 90,000,000 lines of code alone

- Two ways to improve our organization of the memory/data

- QUESTION: Describe what each of these things is in your own words
  - Challenge (perhaps): don’t use programming verbage
Collections of Data

- Rarely do you want to store just one of some value
  - Example: ATLAS pixel detector has ~90,000,000 channels - 90,000,000 separate floats for the energy in each for every events → that would be 90,000,000 lines of code alone

- Two ways to contain collections of things (of the same type) with a single name
  - These are not the only ways to organize data, but very common

**ARRAY**
Fixed number of things
“Static”

**VECTOR**
Variable number of things
“Dynamic”
Arrays

- Two ways to contain collections of things (of the same type) with a single name
  - Arrays: fixed length number of entries of a given type
  - Vectors: expandable via some method calls
- Can roughly treat it as a vector from linear algebra or a “list” of things
  - Accessed with the “index” of the entry

```
myData( int[3] )
```

```
5 ?????? 8
```

```
myData
```

```
5 8 9
```

“Int” amount of data set aside (allocated) here for each spot
Arrays

- Two ways to contain collections of things (of the same type) with a single name
  - Arrays: fixed length number of entries of a given type
  - Vectors: expandable via some method calls
- Can roughly treat it as a **matrix** from linear algebra (“2D list”)
  - Requires two indices for accessing an element

```cpp
myData2D ( int[3][5] )

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>2</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

myData2D

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>??</td>
<td>??</td>
<td>6</td>
<td>??</td>
</tr>
<tr>
<td>2</td>
<td>??</td>
<td>3</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
```
Vectors

- Two ways to contain collections of things (of the same type) with a single name
  - Arrays: fixed length number of entries of a given type
  - Vectors: expandable via some method calls
- Can roughly treat it as a \textit{vector} from linear algebra

```
myData ( vector<int> )

\begin{array}{ccc}
????? & ????? & ????? \\
5 & ????? & ????? \\
5 & 8 & ????? \\
5 & 8 & 9
\end{array}
```
Vectors

- Two ways to contain collections of things (of the same type) with a single name
  - Arrays: fixed length number of entries of a given type
  - Vectors: expandable via some method calls
- Can roughly treat it as a vector from linear algebra ... but you can expand it!

Whenever int's come, I'm be ready
Vectors

- Two ways to contain collections of things (of the same type) with a single name
  - Arrays: fixed length number of entries of a given type
  - Vectors: expandable via some method calls
- Can roughly treat it as a vector from linear algebra
- But it can do so much more than a simple vector!

Whenever `std::vector<int>`'s come, I'm ready.
So Many Data Structures

- You will rarely use all of these - [http://www.cplusplus.com/reference/stl/](http://www.cplusplus.com/reference/stl/)
  - Mastering the use of “vectors” will get you a long way in HEP data analysis
  - Plan to cover “maps” as an advanced topics - “dictionaries” in python

**Sequence containers:**
- array
- vector
- deque
- forward_list
- list

**Associative containers:**
- set
- multiset
- map
- multimap

**Unordered associative containers:**
- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap
My Own Data Structure

- Build from the basic memory structures to facilitate abstraction and the creation of more complex workflows with sensible names
  - This is one type of an “object” = “organizational working structure built from simpler components”
- A few key concepts about structures
  - They can contain any type of lower level structure (int, double, char ...)
  - They are as big as the sum of their component parts
  - Once defined, they provide a new type, which can be treated with the same rules as others structures

Abstraction = simplification of the memory drawing
Structure Implementation

- Define structures prior to using them
  - Perfect place to use the header more effectively
- During implementation we declare them and create a specific instance
- Access subcomponents via the "."
Structure Implementation

- Define structures prior to using them
  - Perfect place to use the header more effectively
- During implementation we declare them and create a specific instance
- Access members of an instance of the struct via the "." operator
Structure Implementation

- Define structures prior to using them
  - Perfect place to use the header more effectively
- During implementation we declare them and create a specific instance
- Access members of an instance of the struct via the “.” operator

```cpp
struct Date {
    int day, month, year;
}

int main() {
    Date today;
    today.day = 30;
    today.month = 1;
    today.year = 2017;
    cout << "Today is: (dd, mm, yyyy): " << today.day << "-" << today.month << "-" << today.year << endl;
    return 0;
}
```
Structure Implementation

- Define structures prior to using them
  - Perfect place to use the header more effectively
- During implementation we **declare** them and create a specific instance
- Access members of an instance of the struct via the "." operator

```cxx
#include <iostream>
#include <vector>

using namespace std;

struct Date {
  int day, month, year;
};

int main(){
  Date today;
  today.day = 30;
  today.month = 1;
  today.year = 2017;
  cout << "Today is: (" << today.day << "-" << today.month << "-" << today.year << ")
      " << endl;
  return 0;
}
```
Structure

- Structures can even contain structures!
  - Make sure that your code is defined in the right order
  - Order of definitions: top to bottom (makes sense)
- Example: Need to know what a **Date** is before I can tell you about my **Exam**

```
struct Date {
  int day, month, year;
};

struct Exam {
  int room, nStudents;
  struct Date date;
};

int main() {
  Exam myExam;
  myExam.room = 103;
  myExam.nStudents = 13;
  cout << "Exam details: day = " << myExam.date.day << " month = " << myExam.date.month << " year = " << myExam.date.year << endl;
  return 0;
}
```
Structure

- Accessing the different members of these “nested” structures takes a cascading style to the usage of the accessor operator.

```
myExam(Exam)

room(int) nStudents(int)

date(Date)

day month year

myExam.date(Date)

day month year

today.day

30
```
Organization is King

- A more appropriate way to arrange the code using a header file
  - Headers: abstract definitions of structs
  - Source code: implementation of usage of instances of structs

- This is not necessary by any means → forming good habits is key to developing
Recall

I want to make sure that you move in the direction of me not being necessary for you to code!
Self-educating: Code documentation

- Learning how to navigate/read documentation is crucial in order to be independent
  - Example: learning about “std::vector” data structure ➔ cplusplus.com reference
  - Question: How do we reset the vector if it becomes too large?
    - More important when dealing with loops (next time)
- What are you going to google search? : “vector in c++” (this is mine)
**Self-educating : Code documentation**

- Eventually, you see that there are descriptions of things you can do with a vector
  - `.at(#)` was used to access the `#` element of the vector
  - `.clear()` seems promising: most code is hotlinked to each other
Self-educating: Code documentation

- Eventually, you see that there are descriptions of things you can do with a vector
  - Understanding these descriptions requires understanding small pieces of code
  - This is why the exercises above have been useful → **IF YOU DO NOT FOLLOW SOME OF THEM, YOU MUST SAY SOMETHING!**
What we haven’t covered

  ○ Allows for “exotic characters” to be interpreted at compilation
  ○ You will encounter bugs if you copy/paste from PDF documents

● The counterparts to “std::cout” - http://www.cplusplus.com/doc/tutorial/basic_io/
  ○ You should be exploring this yourself from your reading on #include <iostream>

● How operations “actually” work
  ○ Bitwise math - http://academic.evergreen.edu/projects/biophysics/technotes/misc/bin_math.htm
  ○ We can parameterize our ignorance at this level and be effective scientists

● Switches
  ○ http://www.cplusplus.com/forum/beginner/20108/
  ○ An alternative method to direct your program flow than if statements
Memory Allocation: EXTRA/BONUS

We can stop here unless you are curious about what is “actually” happening in the computer when we create a variable. E.g. How does the computer store it?
Memory Allocation

- For the curious coder
  - Not essential for all projects

- Some good resources: always start here
  - http://www-bcf.usc.edu/~dkempe/CS104/08-29.pdf
  - http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html#instructions

“chunks of memory”

```
0  32  64  thing3
thing1  thing2
9847  9932  10001
thing5  thing6  thing7
```

..........................
Memory Allocation

- Some good resources: always start here
  - [http://www-bcf.usc.edu/~dkempe/CS104/08-29.pdf](http://www-bcf.usc.edu/~dkempe/CS104/08-29.pdf)
  - [http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html#instructions](http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html#instructions)

Each spot is capable of storing a 1 or a 0.
Memory Allocation

- Some good resources: always start here
  - http://www-bcf.usc.edu/~dkempe/CS104/08-29.pdf
  - http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html#instructions
Memory Allocation

● Compilation creates machine readable directives
  ○ These directives guide the actual physical processes
● Parameterizing your ignorance as a physicist to anything beyond this is OK
Memory Allocation

- The address is just an integer that is displayed in hexadecimal
- Can convert back to integer using this calculator: Hex Calculator
  - 0x7fff5144a5bc → 140734556841404
- The “unit” of this address is in bytes - 1 byte = 8 bits
Memory Allocation

- The address is just an integer that is displayed in hexadecimal
- Can convert back to integer using this calculator: Hex Calculator
  - 0x7fff5144a5bc → 14073456841404
  - 0x7fff514395bc → 14073456771772
- These two spots in memory effectively have nothing to do with each other
Memory with structs

- A struct is as big as its component parts
  - Date = int day + int month + int year
- How will the compiler allocate the memory?
  - Option 1: in an ordered way
  - Option 2: in an unordered way
Memory with structs

- Memory is allocated upon declaring an instance of the struct
  - But nothing is done to this memory → it has whatever 1’s and 0’s it previously had
- Questions:
  - What do you notice about the locations of the memory?
  - Does the location of memory change as we proceed through the program?