A few comments

Your Work area

• Your Work area is not active by default
  ‣ Activate your Work area following these links: cern.ch/it-div —> Account Management —> My Account —> Services… —> AFS Workspaces —> Settings

Issues with computing infrastructure at CERN

• Contact the IT Service Desk at service-desk@cern.ch
  ‣ A ticket will be created and the Service Desk will contact you.
  ‣ They work 9am - 5pm CERN time
Program Layout

- Breaking down our simple program from last time: build up from here
Program Layout

- Breaking down our simple program from last time: build up from here

This is a comment. They can (and should) go throughout the program.
Program Layout

- Breaking down our simple program from last time: build up from here

Bring in other functionality via the inclusion of headers and libraries (Homework 1 …)

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello CERN" << endl;
    return 0;
}
```
Program Layout

- Breaking down our simple program from last time: build up from here

This says that your main function has access to the "std" (standard) namespace
**Program Layout**

- Breaking down our simple program from last time: build up from here

Return value type. Must be an "int" for main()
Program Layout

- Breaking down our simple program from last time: build up from here

How to print stuff to the screen - more options to be learned on your own
Program Layout

- Breaking down our simple program from last time: build up from here

Whitespace - it actually means nothing in C++. Just used for structure/readability
Program Layout

- Breaking down our simple program from last time: build up from here

Return value for the function when it terminates here → can be accessed in terminal afterwards via “?”
Running this program

- Compiling and executing dumps the statement to the terminal as expected
- Able to access the return value of our program after running via “?” in terminal
  - Very useful when program becomes long and can have many endpoints

**QUESTION**: What is this thing called in the shell?
Factorizing : Headers

- If we kept developing everything in one file (session2.cxx) it would become long
  - My first analysis was like this and turned into ~10000 lines of code
- Factorize initially via use of headers - Why? ([Description here](#))
  - Organization and interface vs. implementation: staying organized is essential
  - Re-compilation time: not a big issue now
- Headers are like “ingredient list” or “tool box”
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) \( \rightarrow \) 8*10^9 spaces for either 0 or 1

● Example: integers
  ○ What is 14 in binary? \( 14 < 16 (=2^4) \) \( \rightarrow \) 8+4+2 \( \rightarrow \) 1*2^3+1*2^2+1*2^1+0*2^0 \( \rightarrow \) 1110
  ○ Higher number = more memory
    ■ “maximum integer” of 2147483647 = 2^31-1
    ■ (31 bits for size) + (1 bit for \( \pm \))

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

Convert letters into blocks of bits “byte”

\[ \text{I like to ski in the winter} \]

\[ 01001000 \ 00110101 \ 01000000 \ 01000001 \ 01010010 \ 00000000 \ 00101101 \ 01100001 \ 01100110 \]
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) \(\rightarrow 8 \times 10^9\) spaces for either 0 or 1

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

Spelling: 26 letters + space \(\rightarrow 5\) 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
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? : $14 < 16 (=2^4)$ → $8 + 4 + 2 \rightarrow 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \rightarrow 1110$
- Higher number = more memory
  - “maximum integer” of $2147483647 = 2^{31} - 1$
  - (31 bits for size) + (1 bit for ±)

~Concept of “the stack”
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.

Here's a visual representation of the concepts:

```
<table>
<thead>
<tr>
<th>thing1</th>
<th>thing2</th>
<th>thing3</th>
<th>thing4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>5364</td>
</tr>
<tr>
<td>9847</td>
<td>9932</td>
<td>10001</td>
<td></td>
</tr>
</tbody>
</table>

thing5 = location of thing2 = 32
```
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

```
myXvariable(int)
myYvariable(int)
sum(int)
```

Created and initialized in header

```
//this.is.the.header.of.the.program
int myXvariable = 6;
int sum = 8;
```

Created and initialized in source code

```
#define myYvariable = 4;
```

```
int main()
{
    cout << "Hello CERN" << endl;
    int myYvariable = 4;
    cout << "variables: " << myXvariable << " " << myYvariable << endl;
    int sum = myXvariable + myYvariable;
    cout << "The sum is: " << sum << endl;
    return 0;
}
```
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>
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”
So Many Operators

- Many different types of operators are useful in different places
  - Be aware that they exist - [good overview](http://cplusplus.com) 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>-</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>bitwise OR</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>&amp;</td>
<td>logical OR</td>
</tr>
<tr>
<td>15</td>
<td>Assignment-level expressions</td>
<td>-&gt; &lt;&lt;= &amp;= ^= =</td>
<td>assignment / compound assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>16</td>
<td>Sequencing</td>
<td>; ,</td>
<td>conditional operator</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>
```

This is the origin of the name “c++”:
“\[c++ \leftrightarrow c + stuff\]"
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.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    a = 10;
    b = 4;
    cout << "a = " << a << " b = " << b << endl;
    cout << "c = a + b = " << a + b << endl;
    return 0;
}
```

(L7) ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```

(L8) ```
```cpp
10
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```

(L9) ```
```cpp
10
``` || ```
```cpp
4
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```

(L10) ```
```cpp
14
``` || ```
```cpp
4
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```

(L11) ```
```cpp
14
``` || ```
```cpp
7
``` || ```
```cpp
?????
``` || ```
```cpp
?????
``` || ```
```cpp
?????
```
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>
```
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
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] )
```

```
myData
5  ????
```

```
myData
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

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

0 1 2 3 4
0 ?? ?? ?? ?? ??
1 ?? ?? ?? ?? ??
2 ?? ?? ?? ?? ??
```

```
myData2D

?? ?? ?? ?? ??
?? ?? ?? 6 ??
?? ?? 3 ?? ??
```
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
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

myData (vector<int>)

... myData

\[
\begin{array}{cccc}
5 & & & \\
5 & 8 & & \\
5 & 8 & 9 & \\
\end{array}
\]
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 it can do so much more than a simple vector!
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

<table>
<thead>
<tr>
<th>Sequence containers:</th>
<th>Associative containers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>array C++</td>
<td>Array class (class template )</td>
</tr>
<tr>
<td>vector</td>
<td>Vector (class template )</td>
</tr>
<tr>
<td>deque</td>
<td>Double ended queue (class template )</td>
</tr>
<tr>
<td>forward_list C++</td>
<td>Forward list (class template )</td>
</tr>
<tr>
<td>list</td>
<td>List (class template )</td>
</tr>
<tr>
<td></td>
<td>set (class template )</td>
</tr>
<tr>
<td></td>
<td>multiset Multiple-key set (class template )</td>
</tr>
<tr>
<td></td>
<td>map Map (class template )</td>
</tr>
<tr>
<td></td>
<td>multimap Multiple-key map (class template )</td>
</tr>
<tr>
<td>Container adaptors:</td>
<td>Unordered associative containers:</td>
</tr>
<tr>
<td>stack</td>
<td>LIFO stack (class template )</td>
</tr>
<tr>
<td>queue</td>
<td>FIFO queue (class template )</td>
</tr>
<tr>
<td>priority_queue</td>
<td>Priority queue (class template )</td>
</tr>
<tr>
<td></td>
<td>unordered_set C++ Unordered Set (class template )</td>
</tr>
<tr>
<td></td>
<td>unordered_multiset C++ Unordered Multiset (class template )</td>
</tr>
<tr>
<td></td>
<td>unordered_map C++ Unordered Map (class template )</td>
</tr>
<tr>
<td></td>
<td>unordered_multimap C++ Unordered Multimap (class template )</td>
</tr>
</tbody>
</table>
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 other structures

![Struct Diagram]

```
struct structName {
    memberData1
    memberData2
    memberData3
}
```

“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

```
today(Date)
```
**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
today(Date)
day(int) month(int) year(int)
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>scope qualifier</td>
</tr>
<tr>
<td>++</td>
<td>postfix increment / decrement</td>
</tr>
<tr>
<td>()</td>
<td>functional forms</td>
</tr>
<tr>
<td>[]</td>
<td>subscript</td>
</tr>
<tr>
<td>.</td>
<td>member access</td>
</tr>
</tbody>
</table>

```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

```c++
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

- 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

```cpp
// this is my first C++ program
#include <iostream>
#include <vector>
using namespace std;

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

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

int main() {
    Exam myExam;
    myExam.room = 103;
    myExam.nStudents = 13;
    myExam.date.day = 20;
    myExam.date.month = 1;
    myExam.date.year = 2017;
    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
??       | ??      | ??
```

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

int main() {
    Exam myExam;
    myExam.room = 103;
    myExam.nStudents = 13;
    myExam.date.day = 30;
    myExam.date.month = 3;
    myExam.date.year = 2017;
    cout << "Exam details: day = " << myExam.date.day << 
         " month = " << myExam.date.month << 
         " year = " << myExam.date.year << 
         " room = " << myExam.room << 
         " nStudents = " << myExam.nStudents;
    return 0;
}
```
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
How to compose a workflow

- Code executes from top to bottom
- The rest is tracing the operations and following the path of logic
- And knowing the “scope” in which your code is executing
  - “What happens in vegas stays in vegas”
Write good code!

- Which is bad and why?
Let’s play with a program a bit more complex

The goal

• Let’s try and create a toy program to simulate the interactions between elementary particles (and some not so elementary)
• Explore Structures and Factorization
• First real homework

Here is where the code is. This is a public directory.

```
ls -lrth
```

```
total 18K
-rw-r-xr-x. 1 palacino zp 12K Jan 31 15:02 runHepEx
-rw-r--r--. 1 palacino zp  73 Jan 31 15:02 process.h
-rw-r--r--. 1 palacino zp  729 Jan 31 15:02 process.cxx
-rw-r--r--. 1 palacino zp  399 Jan 31 15:02 particle.h
-rw-r--r--. 1 palacino zp  945 Jan 31 15:02 particle.cxx
-rw-r--r--. 1 palacino zp  1.5K Jan 31 15:02 HepExample.cxx
```

Note that there are two header files and three source files
A toy HEP simulator: particle.h

This is a preprocessor directive to prevent redefinition of variables

```
#ifndef PARTICLE_H
#define PARTICLE_H

struct particle{
    // Use natural units only c=\hbar=k_B=1
    // Let's stick to MeV as our base unit
    double mass;
    int charge;
    double px;
    double py;
    double pz;
    double e;
};

void printParticle(particle);

// Fundamental symmetries
void Charge(particle);
void Parity(particle);

// A relativity check
void isParticleConsistent(particle);
```

A structure to store the information of the particles

Data members of `particle`

Some functions to play with
A toy HEP simulator: particle.cxx

```cpp
#include <iostream>
#include "particle.h"

using namespace std;

void Charge(particle p){
    p.charge *= -1;
    return;
}

void Parity(particle p){
    p.px *= -1;
    p.py *= -1;
    p.pz *= -1;
    return;
}

void isParticleConsistent(particle p){
    double e2 = p.e*p.e + p.px*p.px + p.py*p.py + p.pz*p.pz;
    if(e2 - p.mass*p.mass == 0) cout<<"This is a time-like particle of mass "<e>p.mass<<"."<endl;
    else cout<<"This is a space-like particle of mass "<e>p.mass<<"."<endl;
    return;
}

void printParticle(particle p){
    cout<<"-------- Particle Properties --------"<endl;
    cout<<"mass\t<e>p.mass</e> "<endl;
    cout<<"charge\t<e>p.charge</e> "<endl;
    cout<<"energy\t<e>p.energy</e> "<endl;
    cout<<"momentum-x\t<e>p.px</e> "<endl;
    cout<<"momentum-y\t<e>p.py</e> "<endl;
    cout<<"momentum-z\t<e>p.pz</e> "<endl;
    return;
}
```

Header files and libraries

This source file is where all the functions defined in the header file are implemented.
A toy HEP simulator: process.h

Note that `particle.h` is included here too. This could be problematic. Thus this was added to `particle.h`
cmath library gives access to many math functions and tools

```cpp
#include "process.h"
#include <cmath>

void elasticCollision1D(particle p1, particle p2){
    //create a copy of the initial particles
    particle p1_mod = p1;
    particle p2_mod = p2;

    //for simplicity I'm setting y and z components of the momentum to zero and recalculating the energy
    p1_mod.py = 0; p1_mod.pz = 0;
    p1_mod.e = sqrt(p1_mod.mass*p1_mod.mass + p1_mod.px*p1_mod.px);
    p2_mod.py = 0; p2_mod.pz = 0;
    p2_mod.e = sqrt(p2_mod.mass*p2_mod.mass + p2_mod.px*p2_mod.px);

    // From (P_1 + P_2)^2 = (P_3 + P_4)^2 --> e1*e2 - pX1*pX2 = e3*e4 pX3*pX4, and e = sqrt(p^2-m^2)
    // From pX1 + pX2 = pX3 + pX4

    // Solve the system of equations here

    p1 = p1_mod;
    p2 = p2_mod;

    return;
}
```

Finishing the development of this will be your homework
In principle one should avoid including headers multiple times.

This is the `main()` that steers the whole program.

Exit code 0. This is the most common value.
• Copy the code in /afs/cern.ch/work/p/palacino/public/forCSUStudents/HepExample over to your area

Part 1
• Create a new structure and a function that takes the new structure as an argument
  ‣ You can do this in particle, process or in a new header+source file
• Use that function in HepExample.cxx

Part 2
• Solve the system of equations in process.cxx and finish the implementation of elasticCollision(). Use ir in HepExample.cxx.

Part 3
• Create a tarball with the header and the source files and email it to me with the subject Programing - Homework 1 - <your name>

Due midnight of Thursday, February 7.