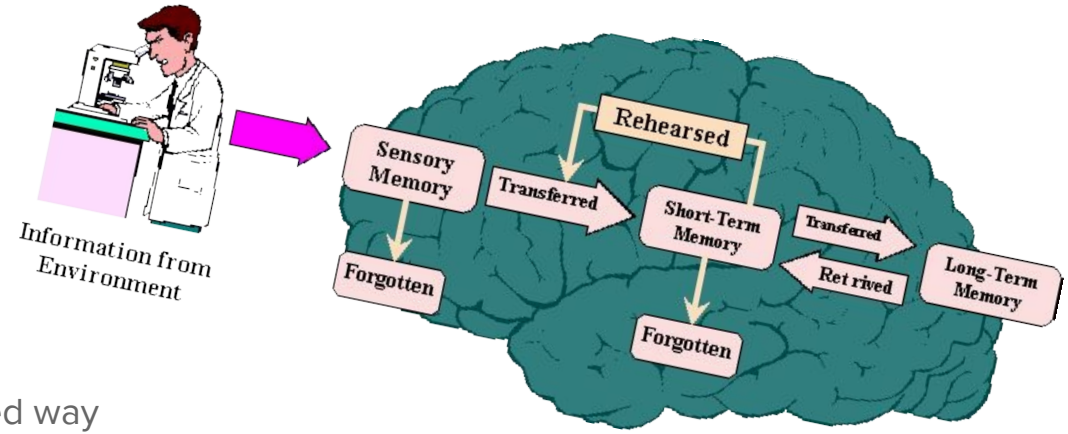
Programming 2 : Memory and Navigation

Sam Meehan
University of Washington
31 January 2017

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

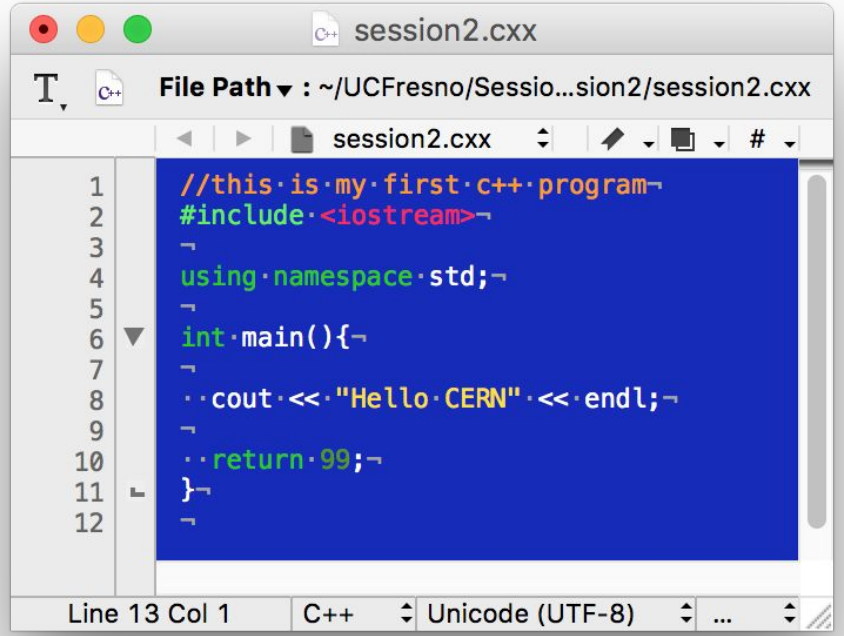
Introduction



- Program layout
  - Generic starting point
- Computer memory :
  - How to think about it
  - How to use it in a basic way
  - How to use it in a more advanced way
- Program flow control : “conditionals”
  - Designing/formatting them effectively
- 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”

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.

"main()" function declaration and definition → This is the place to start reading the code to understand what it will be doing.

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

Adding arguments to the main function is possible

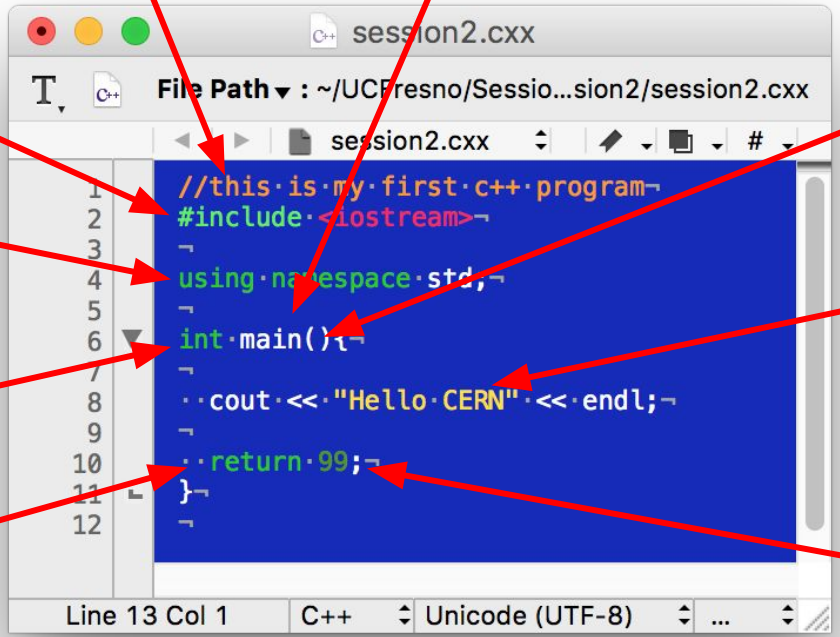
This says that your main function has access to the "std" (standard) **namespace**

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

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

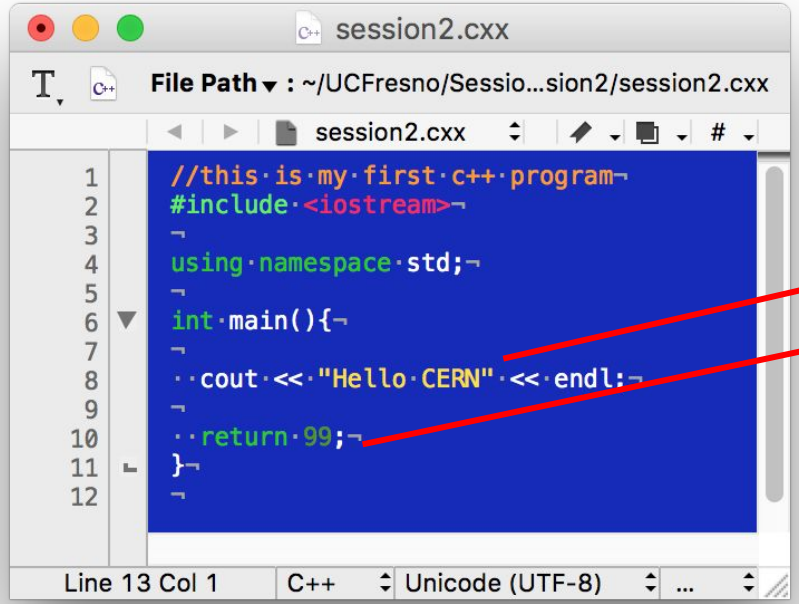
Whitespace - it actually means nothing in c++. Just used for structure/readability

Return value for the function when it terminates here → can be accessed in terminal afterwards via "?"

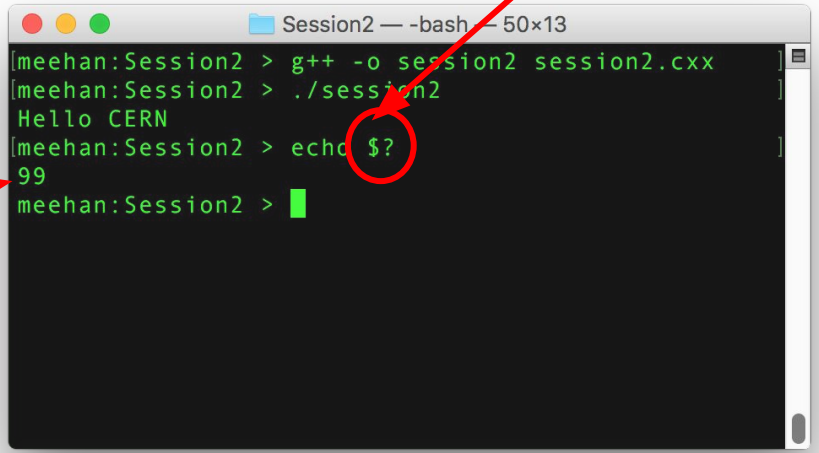


Running this program

- 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

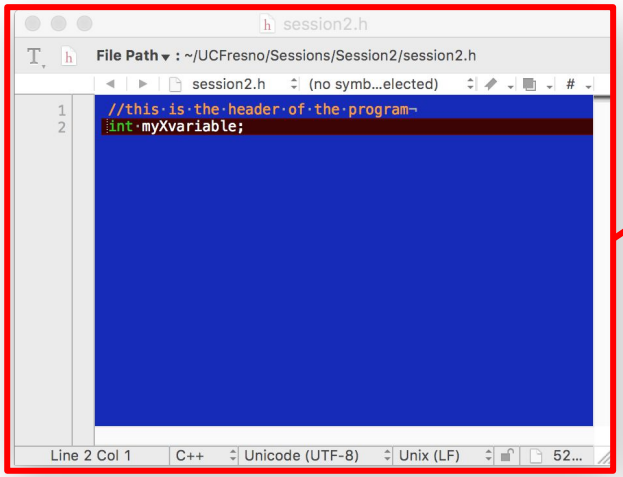


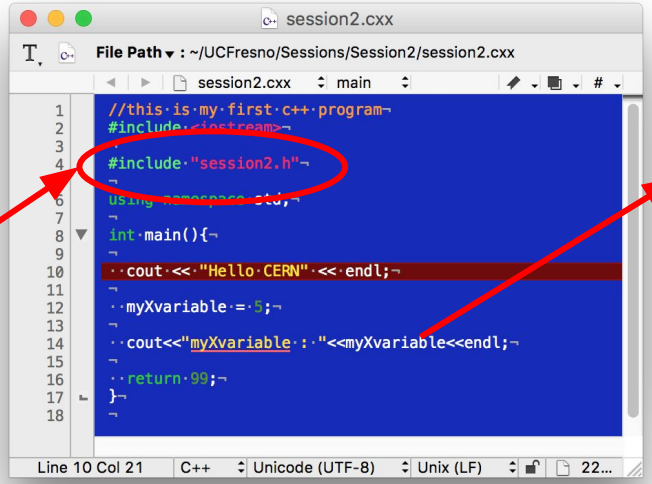
QUESTION : What is this thing called in the shell?

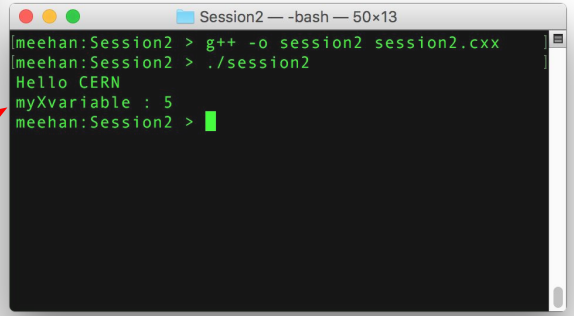


Factorizing : Headers

- 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 “ingedient 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) ➔ $8*10^9$ spaces for either 0 or 1
- Example : integers
  - What is 14 in binary? : $14 < 16(=2^4)$ ➔ $8+4+2$ ➔ $1*2^3+1*2^2+1*2^1+0*2^0$ ➔ 1 1 1 0
  - Higher number = more memory
    - “maximum 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 →

01001000000011010100101100001010000010101100000
00001010001100010010000001001011110000010101010
00000101000001100001001011111010100101100 11

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? : $14 < 16(=2^4)$ ➔ $8+4+2$ ➔ $1*2^3+1*2^2+1*2^1+0*2^0$ ➔ 1 1 1 0
  - Higher number = more memory
    - “maximum 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 ➔

01001000000110101001011000010100000101011000000000101000011000100100000010010111100000101010100000101000001100001001011111010100101100110000010100001100010010000001001011110000010101010000010100000110000100101111101010010110011

Small corruption produces gibberish

10100100000011010100101100001010000010101100000000010100001100010010000001001011110000010101010000010100000110000100101111101010010110011 ➔ tpftvbpjx jfdpdwpjtbp dwzry

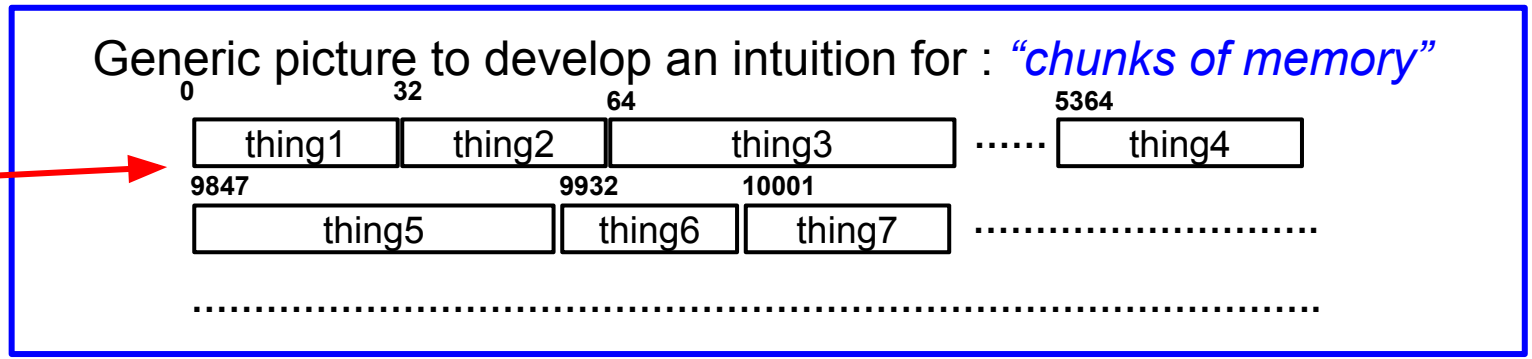
Memory

- 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? : $14 < 16(=2^4)$ → $8+4+2$ → $1*2^3+1*2^2+1*2^1+0*2^0$ →

1	1	1	0

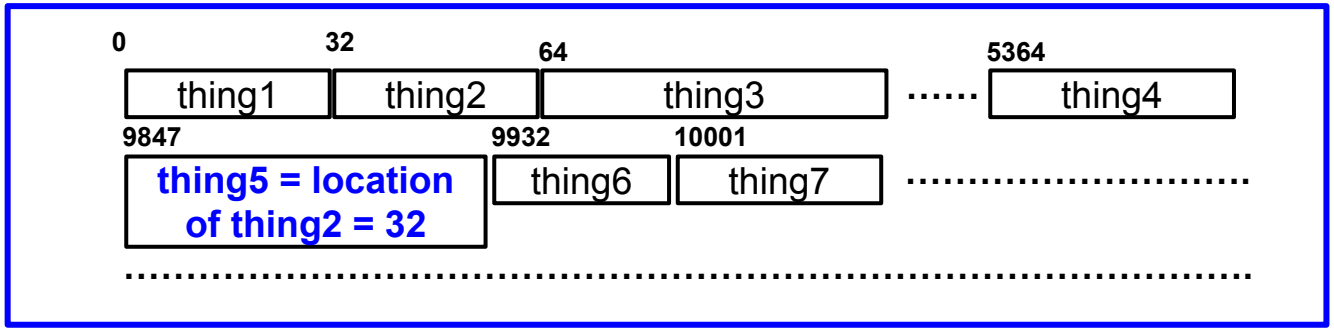
  - Higher number = more memory
    - “maximum integer” of 2147483647 = $2^{31}-1$
    - (31 bits for size) + (1 bit for ±)

~Concept of “the stack”



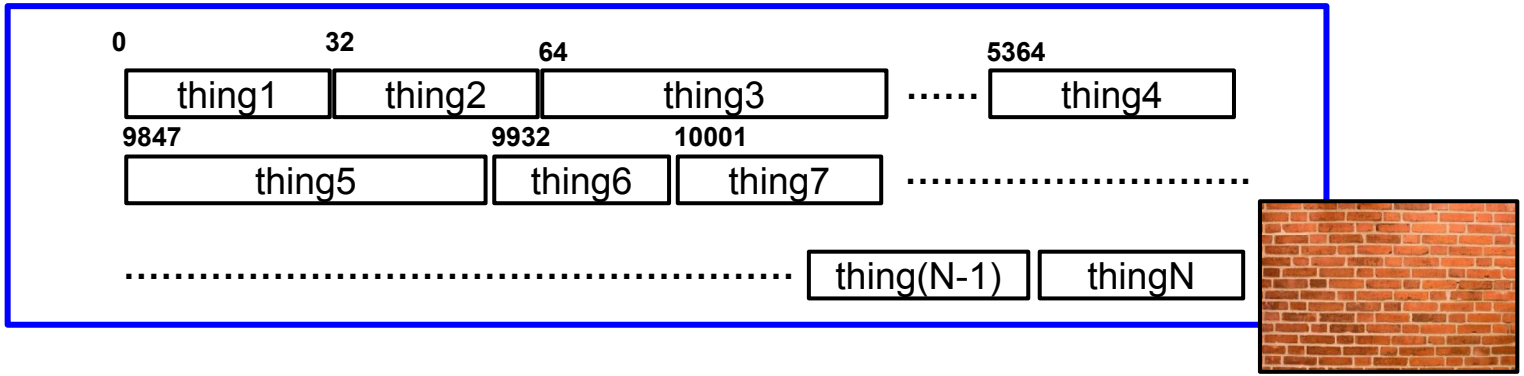
Memory - Funny Concepts

- 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



Memory - Funny Concepts

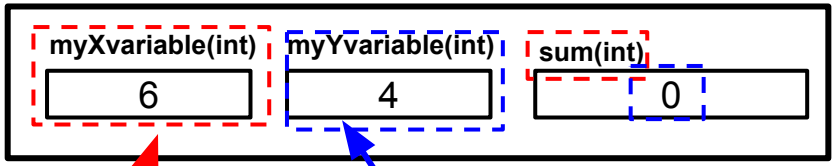
- Two interesting concepts concerning memory result from this picture
  - “Pointers” : the thing stored in memory tells you the location of another bit of memory
  - “Memory Leak” : if your program somehow just keeps assigning memory to new “things” in an uncontrolled way



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”
  - ***initilization*** : what the initial value is that is stored in the place in memory



Created and initialized in header

Created and initialized in source code

session2.h

File Path : ~/UCFresno/Sessions/Session2/session2.h

```
1  //this is the header of the program
2  int myXvariable=6;
3  int sum;
```

Line 3 Col 9 | C++ | Unicode (UTF-8) | Unix (LF) | 63 / 12 / 3

session2.cxx

File Path : ~/UCFresno/Sessions/Session2/session2.cxx

```
1  //this is my first c++ program
2  #include <iostream>
3
4  #include "session2.h"
5
6  using namespace std;
7
8  int main(){
9
10   cout << "Hello CERN" << endl;
11
12   int myYvariable = 4;
13
14   cout<<"variables : "<<myXvariable<<" "<<myYvariable<<endl;
15
16   sum = 0;
17
18   sum = myXvariable + myYvariable;
19
20   cout<<"The sum is : "<<sum<<endl;
21
22   return 99;
23 }
```

Line 16 Col 11 | C++ | Unicode (UTF-8) | Unix (LF) | 331 / 40 / 25

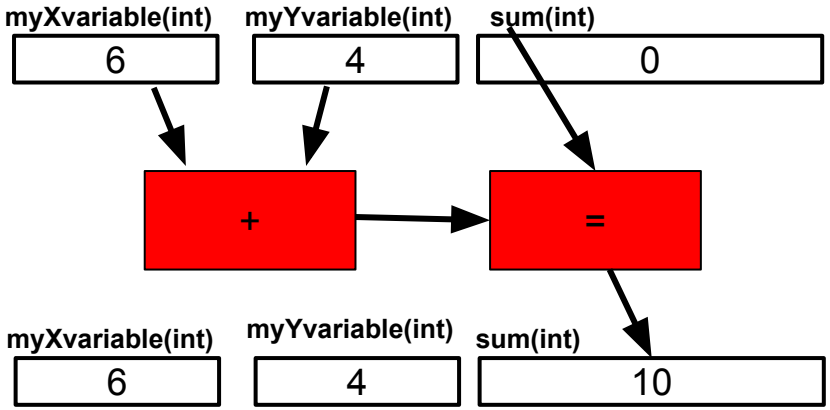
Integers … but there's more

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

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

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



session2.h

File Path : ~/UCFresno/Sessions/Session2/session2.h

```
1  //this is the header of the program
2  int myXvariable=6;
3  int sum;
```

Line 3 Col 9 | C++ | Unicode (UTF-8) | Unix (LF) | 63 / 12 / 3

session2.cxx

File Path : ~/UCFresno/Sessions/Session2/session2.cxx

```
1  //this is my first c++ program
2  #include <iostream>
3
4  #include "session2.h"
5
6  using namespace std;
7
8  int main(){
9
10   cout << "Hello CERN" << endl;
11
12   int myYvariable = 4;
13
14   cout<< variables : "<<myXvariable<<" "<<myYvariable<<endl;
15
16   sum = 0;
17
18   sum = myXvariable + myYvariable;
19
20   cout<<"The sum is : "<<sum<<endl;
21
22   return 99;
23 }
```

Line 16 Col 11 | C++ | Unicode (UTF-8) | Unix (LF) | 331 / 40 / 25

So Many Operators

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

This is the origin of the name “c++” :
“ *c++ ⟷ c + stuff* ”

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

Advanced

Advanced

Advanced

Advanced

Advanced

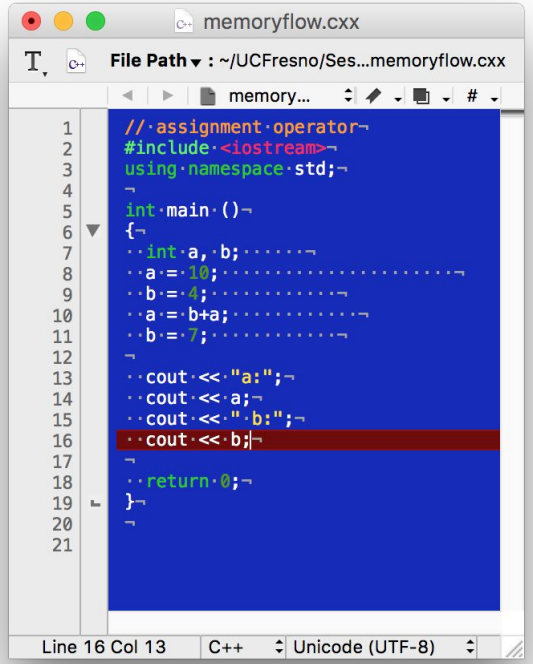
Advanced

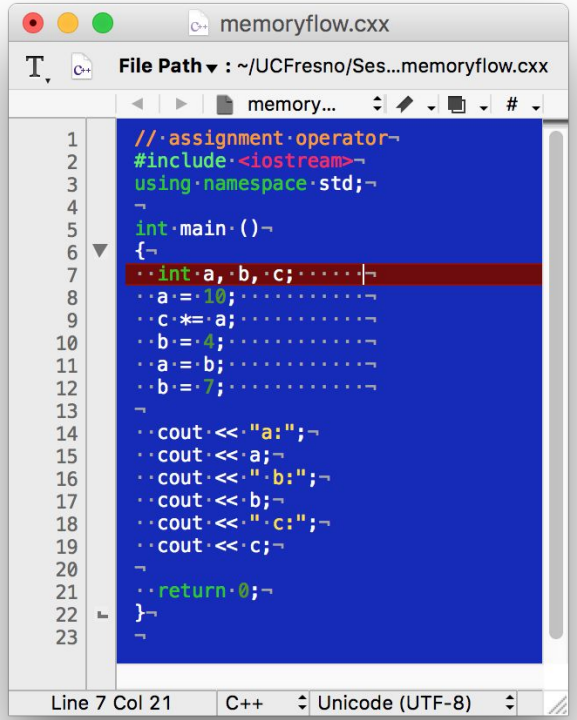
Advanced

Advanced

Exercise : Memory Chart

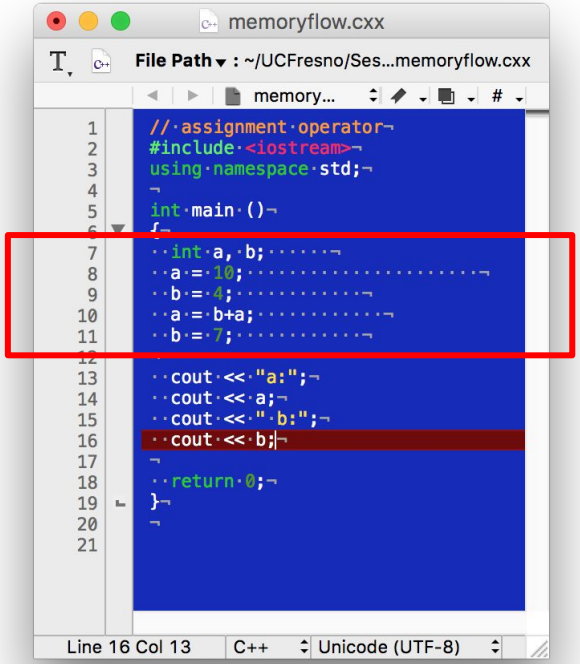
Draw a memory flow on a piece of paper in front of you

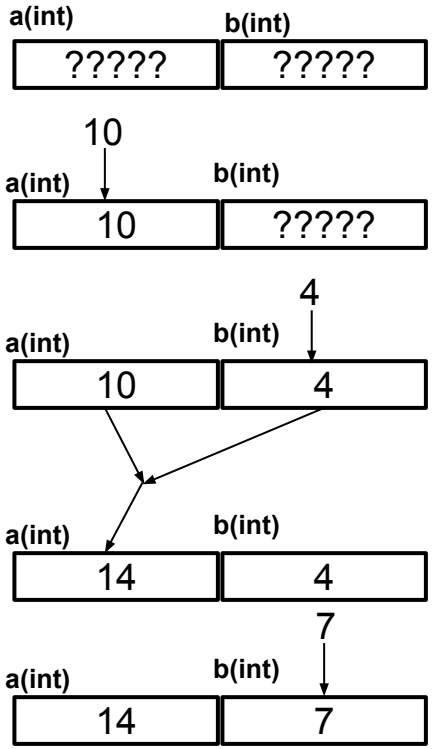




Exercise : Memory Chart

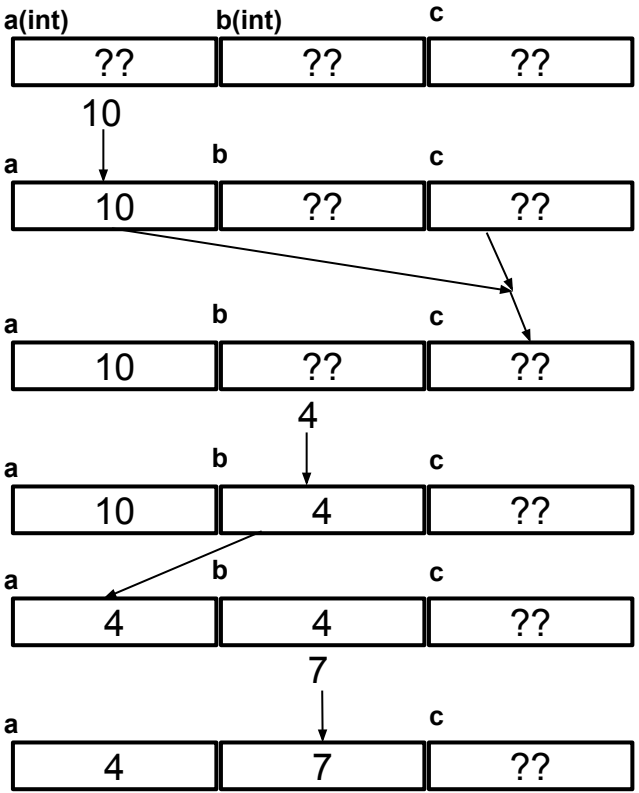
Draw a memory flow on a piece of paper in front of you

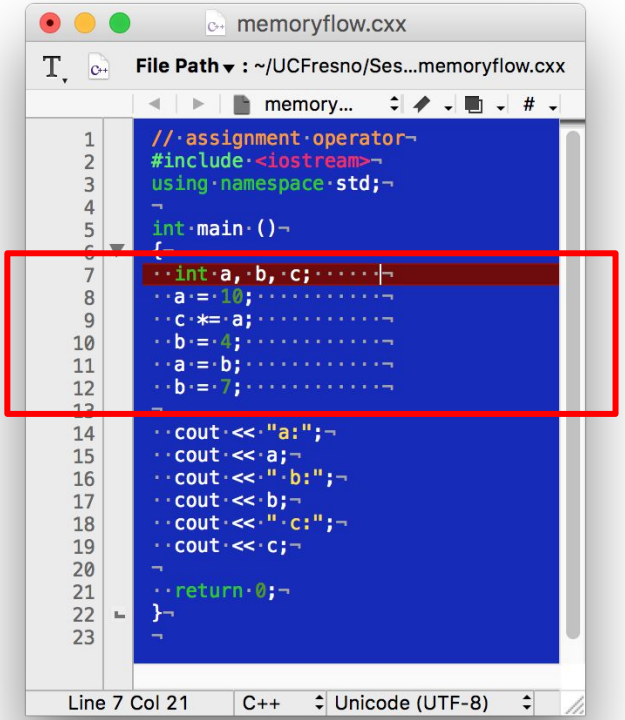




Exercise : Memory Chart

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





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

ARRAY

VECTOR

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"

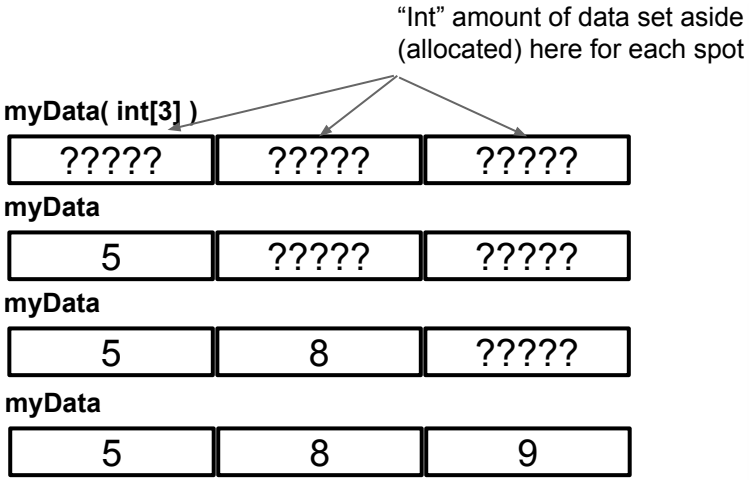
VECTOR

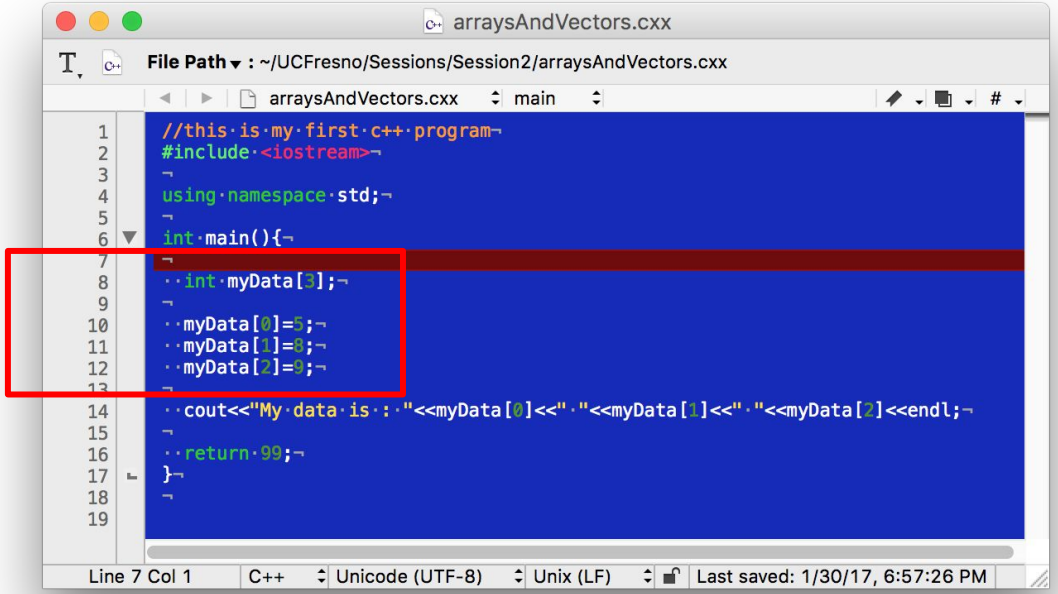
Variable number of things

"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





Arrays

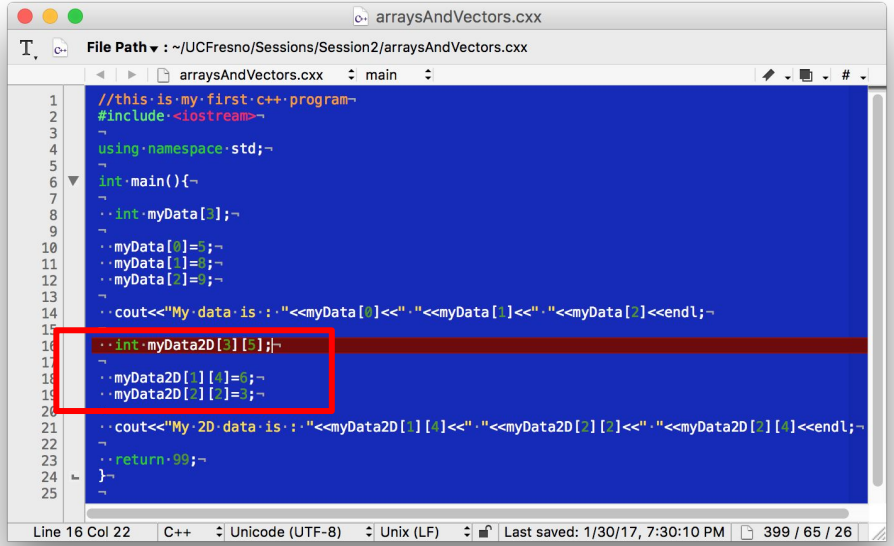
- Two ways to contain collections of things (of the same type) with a single name
  - Arrays : fixed length number of entries of a given type
  - Vectors : expandable via some method calls
- Can roughly treat it as a **matrix** from linear algebra

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

myData (vector<int>)

?????	?????	?????

myData

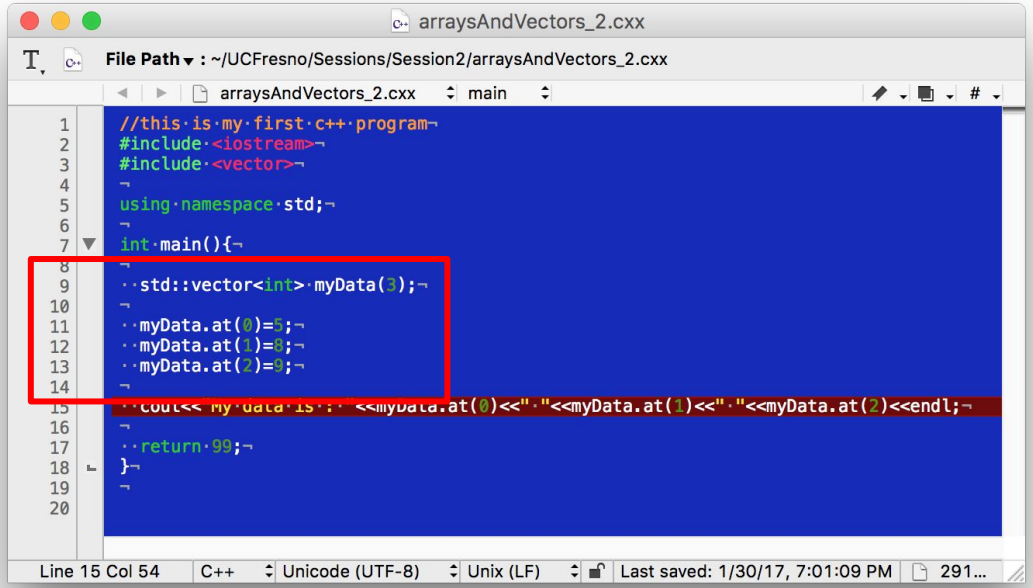
5	?????	?????

myData

5	8	?????

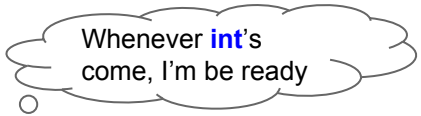
myData

5	8	9



Vectors

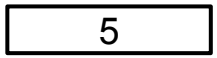
- Two ways to contain collections of things (of the same type) with a single name
  - Arrays : fixed length number of entries of a given type
  - Vectors : expandable via some method calls
- Can roughly treat it as a **vector** from linear algebra



myData (vector<int>)

…

myData

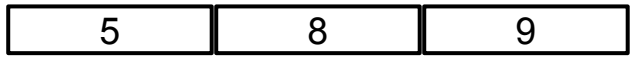


…

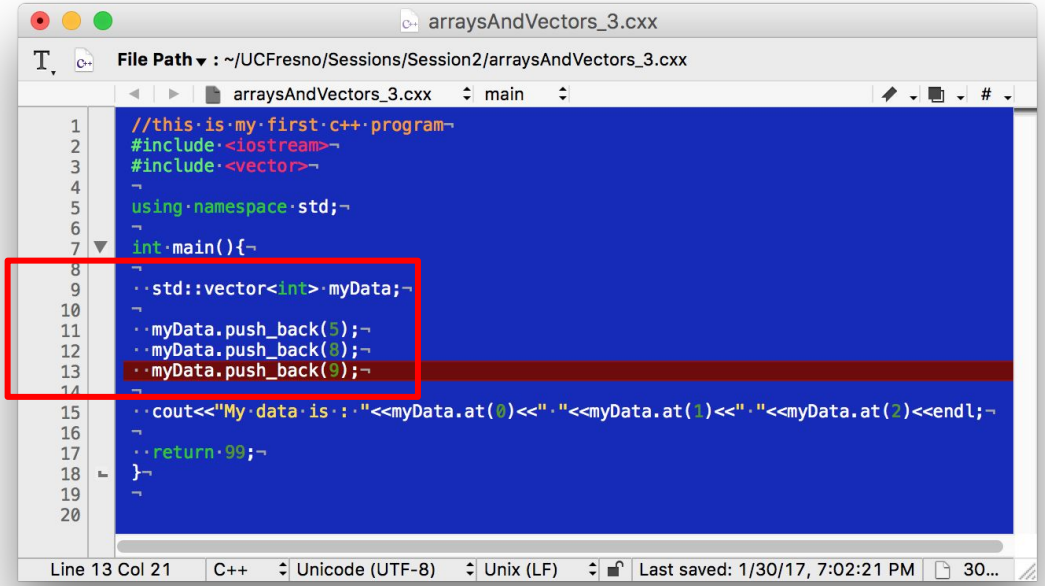
myData

…

myData



…

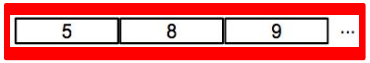


Vectors

- 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 be ready

myData (vector<int>)



myData2D (vector< vector<int> >)

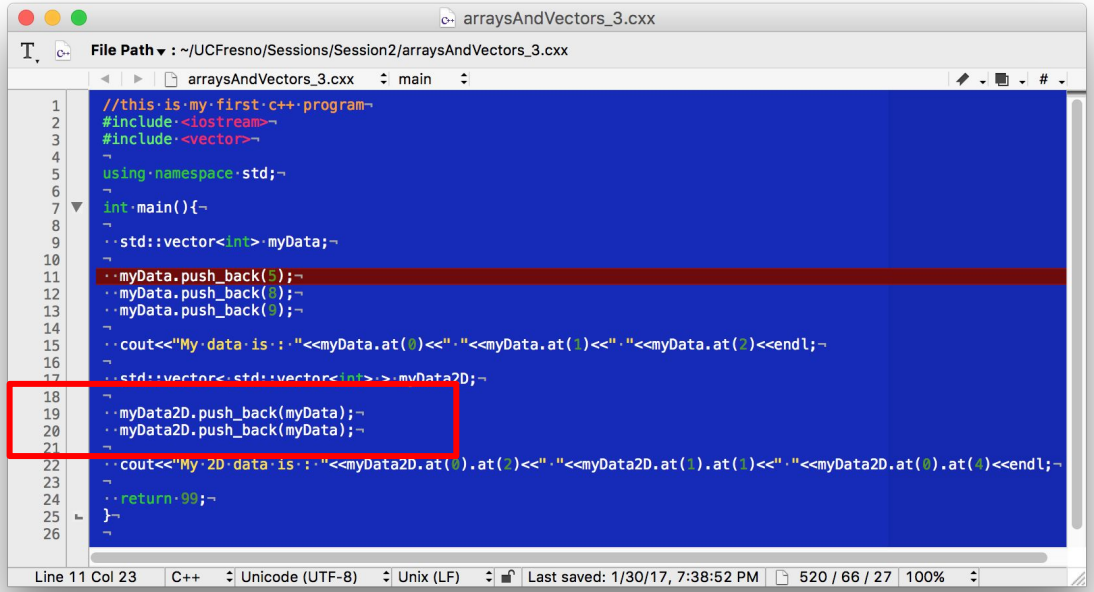
...

myData2D

...

myData2D

...



So Many Data Structures

- You will rarely use all of these - http://www.cplusplus.com/reference/stl/
  - Mastering the use of “vectors” will get you a long way in HEP data analysis
  - Could cover something like “maps” as an advanced topics

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

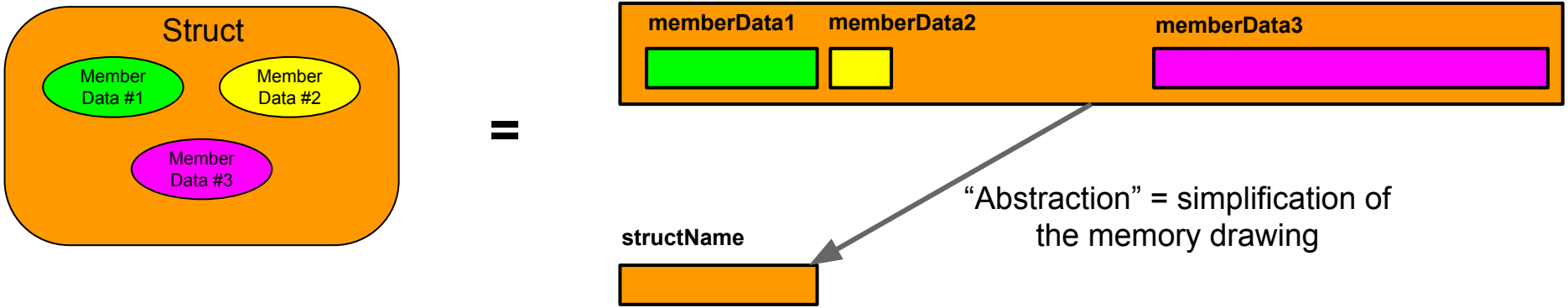
set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

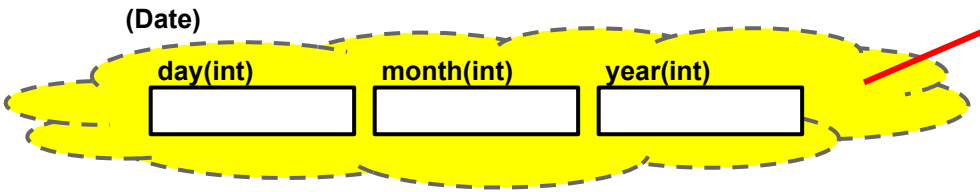
My Own Data Structure

- Build from the basic memory structures to facilitate abstraction and the creation of more complex workflows with sensible names
- Enter the concept of a struct : http://www.cplusplus.com/doc/tutorial/structures/
  - 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



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 “.”



struct.cxx

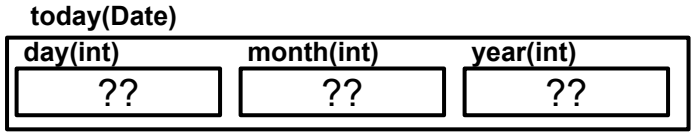
File Path : ~/UCFresno/Sessions/Session2/struct.cxx

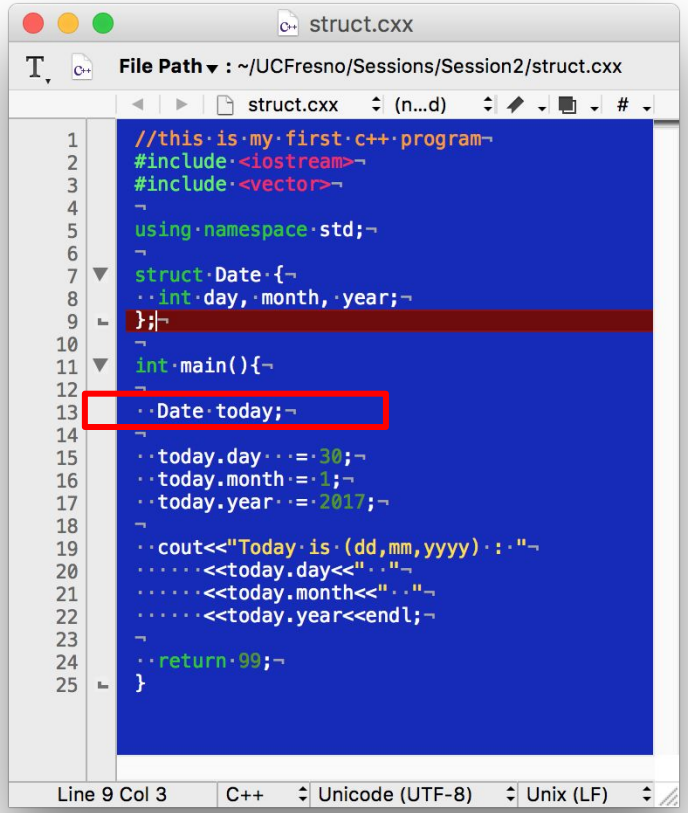
```
1  //this is my first c++ program
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  struct Date {
8    int day, month, year;
9  };
10
11 int main(){
12
13   Date today;
14
15   today.day   = 30;
16   today.month = 1;
17   today.year  = 2017;
18
19   cout<<"Today is (dd,mm,yyyy) :"
20       <<today.day<<"."
21       <<today.month<<"."
22       <<today.year<<endl;
23
24   return 99;
25 }
```

Line 9 Col 3 | C++ | Unicode (UTF-8) | Unix (LF)

Structure Implementation

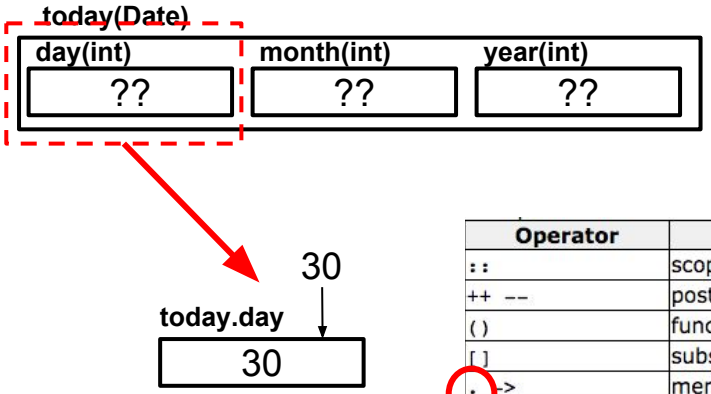
- Define structures prior to using them
  - Perfect place to use the header more effectively
- During implementation we declare them and create a specific instance
- Access members of an instance of the struct via the "." operator





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



Operator	Description
::	scope qualifier
++ --	postfix increment / decrement
( )	functional forms
[ ]	subscript
. ->	member access

struct.cxx

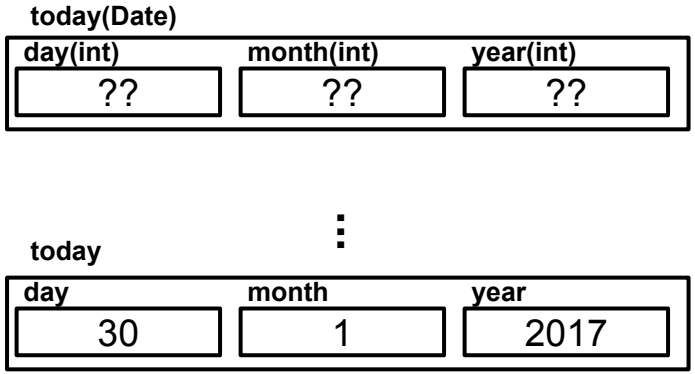
File Path : ~/UCFresno/Sessions/Session2/struct.cxx

```
1  //this is my first c++ program
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  struct Date {
8    int day, month, year;
9  };
10
11 int main(){
12
13   Date today;
14
15   today.day   = 30;
16   today.month = 1;
17   today.year  = 2017;
18
19   cout<<"Today is (dd,mm,yyyy) : "
20       <<today.day<<" "
21       <<today.month<<" "
22       <<today.year<<endl;
23
24   return 99;
25 }
```

Line 9 Col 3 | C++ | Unicode (UTF-8) | Unix (LF)

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

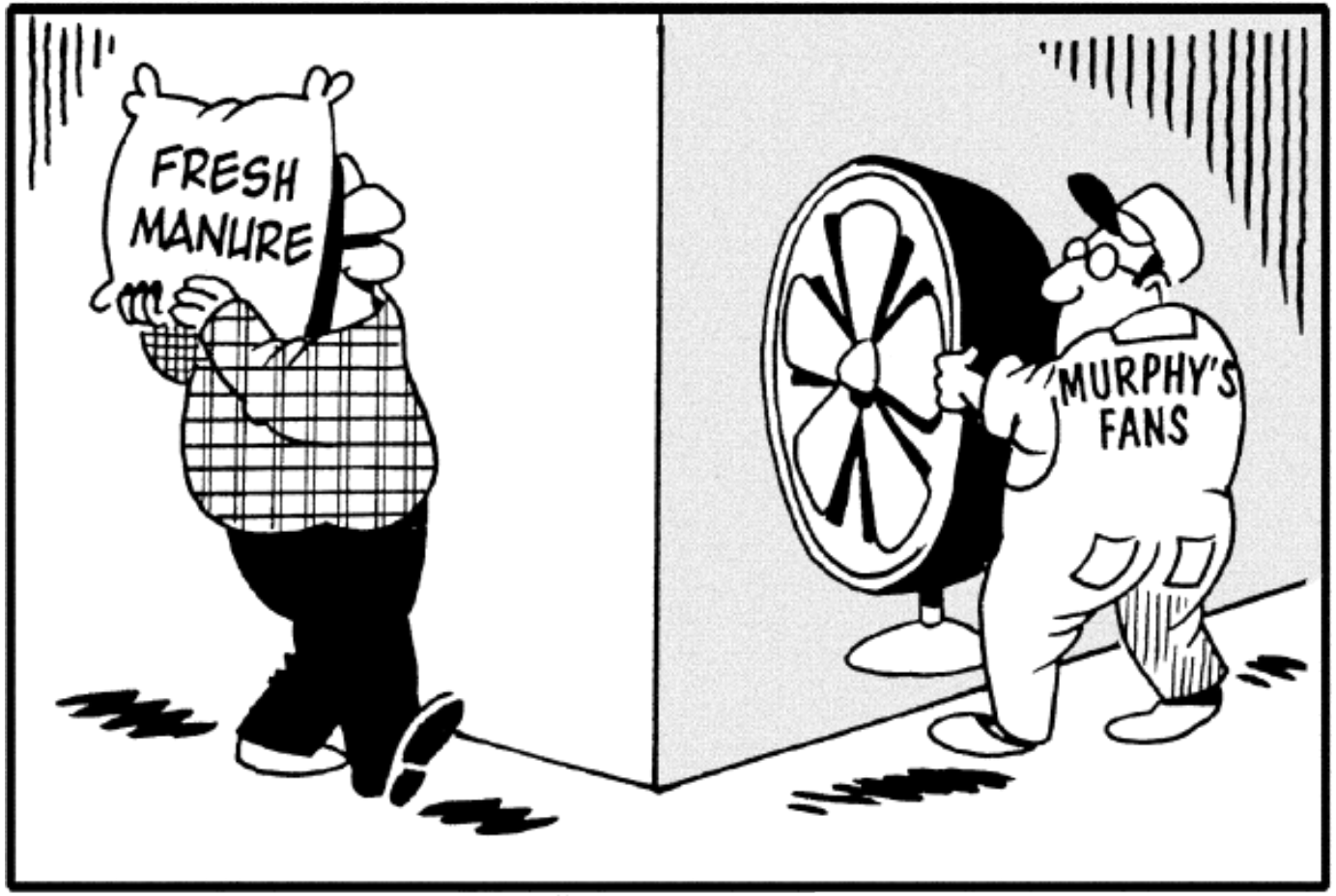


struct.cxx

File Path : ~/UCFresno/Sessions/Session2/struct.cxx

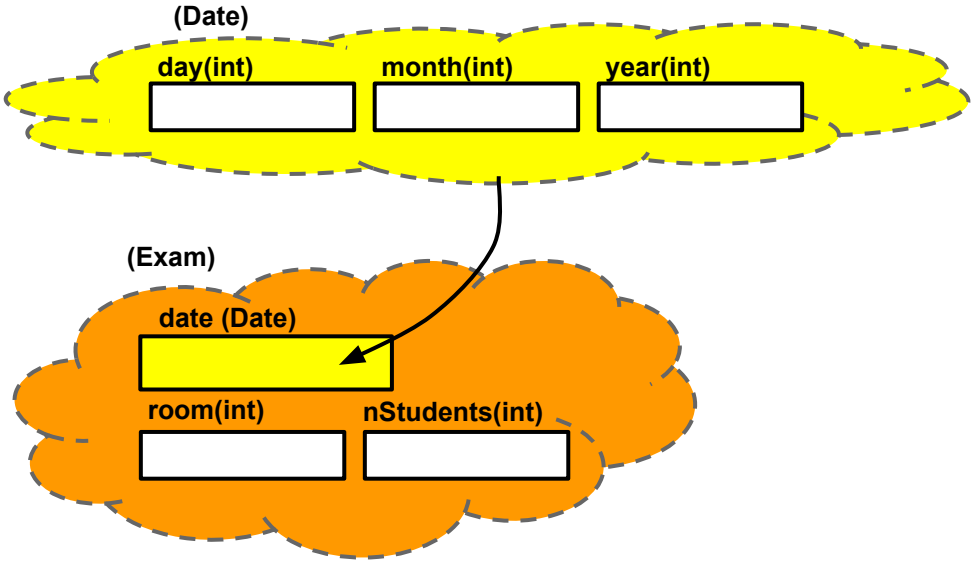
```
1  //this is my first c++ program
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  struct Date {
8    int day, month, year;
9  };
10
11 int main(){
12
13   Date today;
14
15   today.day   = 30;
16   today.month = 1;
17   today.year  = 2017;
18
19   cout<<"Today is (dd,mm,yyyy) : "
20       <<today.day<<" "
21       <<today.month<<" "
22       <<today.year<<endl;
23
24   return 99;
25 }
```

Line 9 Col 3 | C++ | Unicode (UTF-8) | Unix (LF)


FRESH MANURE
MURPHY'S FANS

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**



struct2.cxx

File Path : ~/UCFresno/Sessions/Session2/struct2.cxx

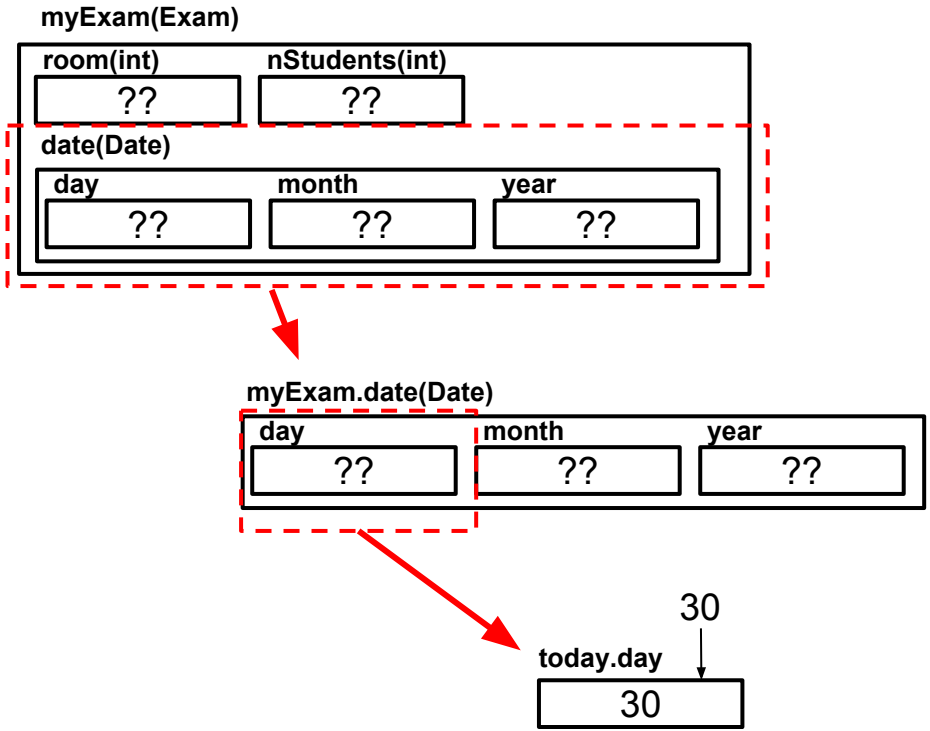
struct2.cxx | main

```
1  //this is my first c++ program
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  struct Date {
8    int day, month, year; };
9
10 struct Exam {
11   int room, nStudents;
12   struct Date date;
13 };
14
15 int main(){
16
17   Exam myExam;
18
19   myExam.room = 103;
20   myExam.nStudents = 13;
21
22   myExam.date.day   = 30;
23   myExam.date.month = 1;
24   myExam.date.year  = 2017;
25
26   cout<<"Exam details : day = "<<myExam.date.day
27       <<"  month = "<<myExam.date.month
28       <<"  year = "<<myExam.date.year
29       <<"  room = "<<myExam.room
30       <<"  nstudents = "<<myExam.nStudents
31       <<endl;
32
33   return 99;
34 }
35
```

Line 20 Col 25 | C++ | Unicode (UTF-8) | Unix (LF) | 5...

Structure

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



struct2.cxx

File Path : ~/UCFresno/Sessions/Session2/struct2.cxx

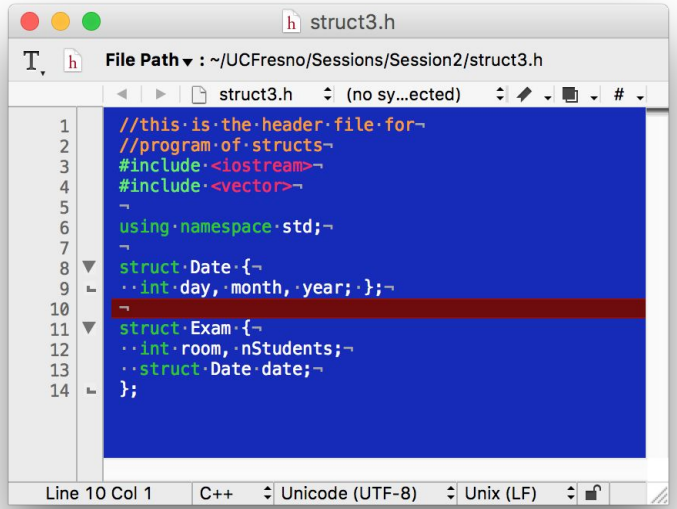
struct2.cxx | main

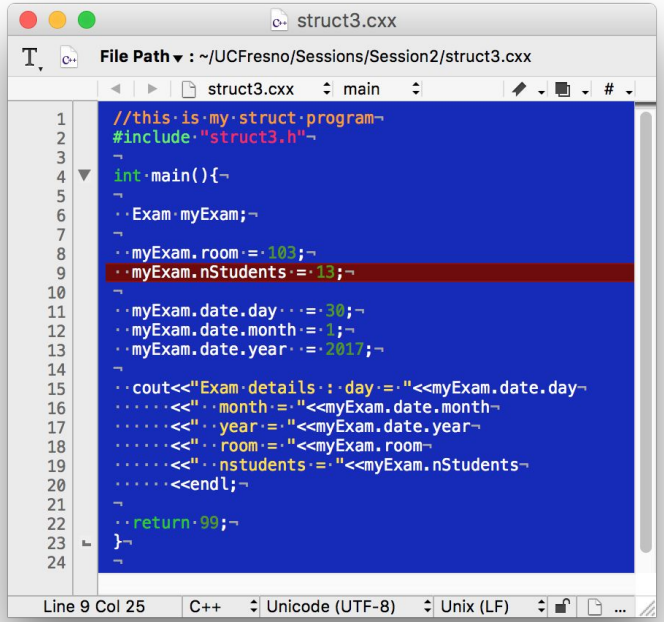
```
1  //this is my first c++ program
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  struct Date {
8    int day, month, year; };
9
10 struct Exam {
11   int room, nStudents;
12   struct Date date;
13 };
14
15 int main(){
16
17   Exam myExam;
18
19   myExam.room = 103;
20   myExam.nStudents = 13;
21
22   myExam.date.day   = 30;
23   myExam.date.month = 1;
24   myExam.date.year  = 2017;
25
26   cout<<"Exam details : day = "<<myExam.date.day
27       <<"  month = "<<myExam.date.month
28       <<"  year = "<<myExam.date.year
29       <<"  room = "<<myExam.room
30       <<"  nstudents = "<<myExam.nStudents
31       <<endl;
32
33   return 99;
34 }
35
```

Line 20 Col 25 | C++ | Unicode (UTF-8) | Unix (LF)

Organization is King

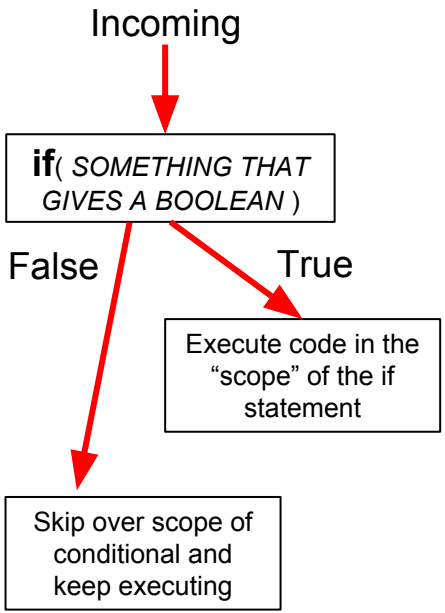
- A more appropriate way to arrange the code using a header
  - Headers : abstract definitions of structs
  - Source code : implementation of usage of instances of structs
- This is not necessary by any means ➡ forming good habits is key to developing





How to compose a workflow

- Direct flow of execution using conditional statements - “if statements”
  - True = anything that is *not false* / False = 0 or “null”



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

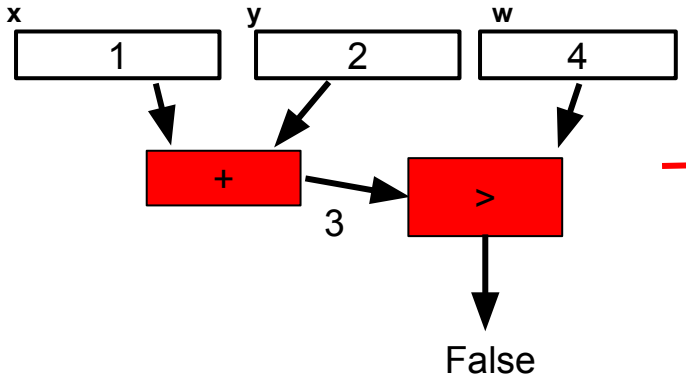
Negate whatever result you previously had

“Asking a question” :
Follow your math intuition
(3>5) → false
(3<5) → true

Truth Tables :
If I have an orange OR I have ice cream, then I am happy.
I have an orange → ???

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”



workflow2bad.cxx

File Path : ~/UCFresno/Sessions/Session2/workflow2bad.cxx

```
1  //this is my first workflow
2  #include <iostream>
3
4  using namespace std;
5
6  int main(){
7
8      int x=1;
9      int y=2;
10     int w=4;
11
12     if( x>y )
13         cout<<"x is greater than y"<<endl;
14
15     if( y==3 || y==w){
16         cout<<"y is equal to 3 or w, I don't know which"<<endl;
17         if( y==3 )
18             cout<<"got it, y is 3"<<endl;
19         else
20             cout<<"y must be w, or I wouldn't be here"<<endl;
21
22         int z=6;
23         cout<<"z value :"<<z<<endl;
24     }
25
26     if( !( w > (x+y) ) )
27         cout<<"w is not greater than the sum of x and y"<<endl;
28
29     if( y=3 )
30         cout<<"This statement was true"<<endl;
31
32     if( x<y && w>x*y )
33         cout<<"Will this print?"<<endl;
34
35     return 99;
36 }
```

Line 24 Col 4 | C++ | Unicode (UTF-8) | Unix (LF) | 633 / 118 / 38

Write good code!

- Which is bad and why?

workflow2bad.cxx

File Path : ~/UCFresno/Sessions/Session2/workflow2bad.cxx

workflow2bad.cxx | main

```
1  //this is my first workflow
2  #include <iostream>
3
4  using namespace std;
5
6  int main(){
7
8    int p1[2];
9    int p2[2];
10
11   p1[0]=2;
12   p1[1]=4;
13
14   p2[0]=6;
15   p2[1]=1;
16
17   if( p1[0]*p1[0]+p1[1]*p1[1] > p2[0]*p2[0]+p2[1]*p2[1] ){
18     cout<<"case 1"<<endl;
19
20     if( p1[0]>0 && p1[1]>0 )
21       cout<<"case 1a"<<endl;
22     else if( p1[0]>0 && p1[1]<0 )
23       cout<<"case 1b"<<endl;
24     else if( p1[0]<0 && p2[1]>0 )
25       cout<<"case 1c"<<endl;
26     else
27       cout<<"case 1d"<<endl;
28
29     int x = p1[0]*p1[1];
30
31     cout<<"quantity : "<<x<<endl;
32   }
33   else{
34     cout<<"case 2"<<endl;
35   }
36
37   return 99;
38 }
39
40
```

Line 29 Col 19 | C++ | Unicode (UTF-8) | Unix (LF) | 585 / 108 / 40

workflow2.cxx

File Path : ~/UCFresno/Sessions/Session2/workflow2.cxx

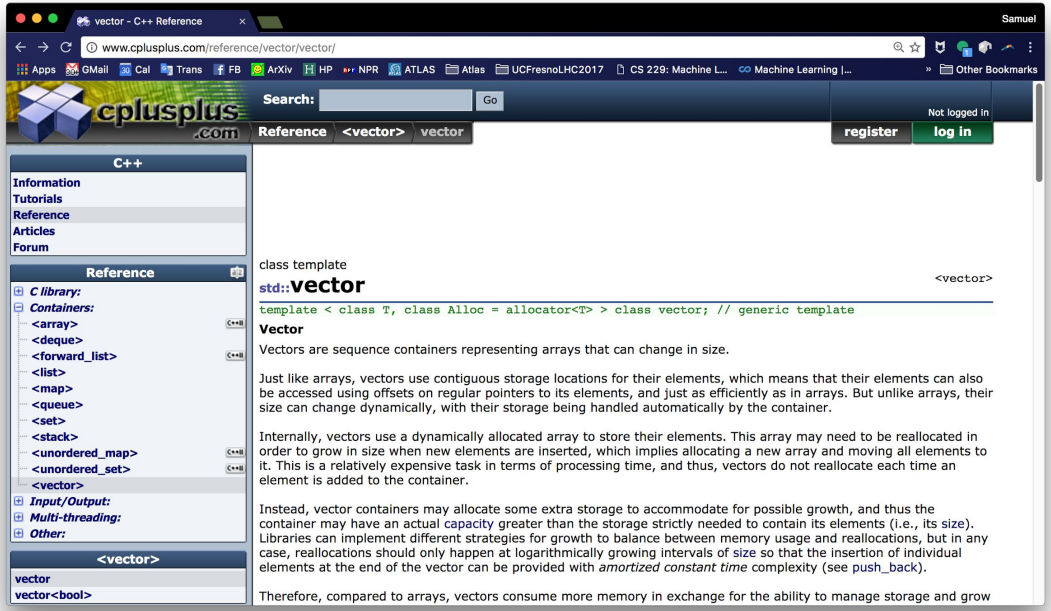
workflow2.cxx | main

```
1  //Description : This takes two points and will analyze point 1
2  //if it is further from the origin.  It will then tell you which
3  //origin it is located in and the area that is made by the rectangle
4  //with one corner at (x,y)=(0,0) and the other at (x,y)=(p1x,p1y)
5
6  #include <iostream>
7
8  using namespace std;
9
10 int main(){
11
12   int p1[2];
13   int p2[2];
14
15   //initialization of the points p1 and p2
16   p1[0]=2;
17   p1[1]=4;
18
19   p2[0]=6;
20   p2[1]=1;
21
22   cout<<"Initial points : "<<endl
23       <<"p1 : x="<<p1[0]<<"  y="<<p1[1]<<endl
24       <<"p2 : x="<<p2[0]<<"  y="<<p2[1]<<endl<<endl;
25
26   if( p1[0]*p1[0]+p1[1]*p1[1] > p2[0]*p2[0]+p2[1]*p2[1] ){
27     cout<<"p1 is further from the origin than p2"<<endl;
28
29     if( p1[0]>0 && p1[1]>0 )
30       cout<<"p1 is in quadrant 1"<<endl;
31     else if( p1[0]>0 && p1[1]<0 )
32       cout<<"p1 is in quadrant 2"<<endl;
33     else if( p1[0]<0 && p2[1]>0 )
34       cout<<"p1 is in quadrant 3"<<endl;
35     else
36       cout<<"p1 is in quadrant 4"<<endl;
37
38     int areaWithOrigin = p1[0]*p1[1];
39
40     cout<<"p1 distance squared is : "<<areaWithOrigin<<endl;
41   }
42   else{
43     cout<<"p2 is further away -- I don't care about this"<<endl;
44   }
45
46   return 99;
47 }
```

Line 15 Col 43 | C++ | Unicode (UTF-8) | Unix (LF) | 1,155 / 212 / 49

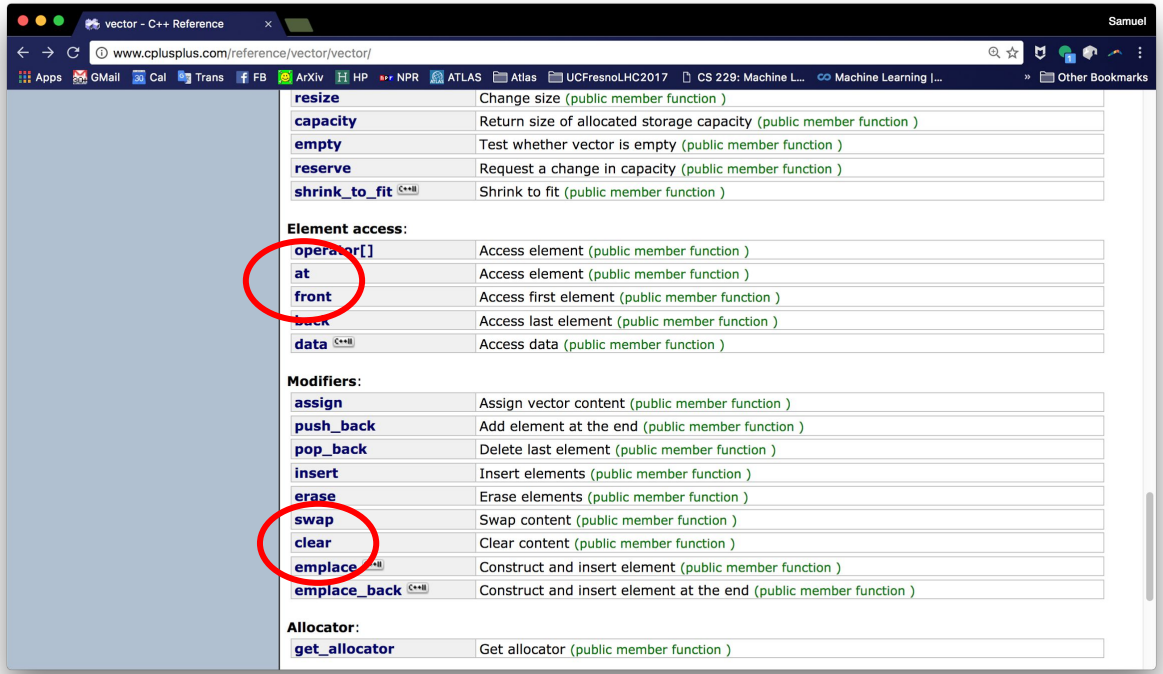
Self-educating : Code documentation

- 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)



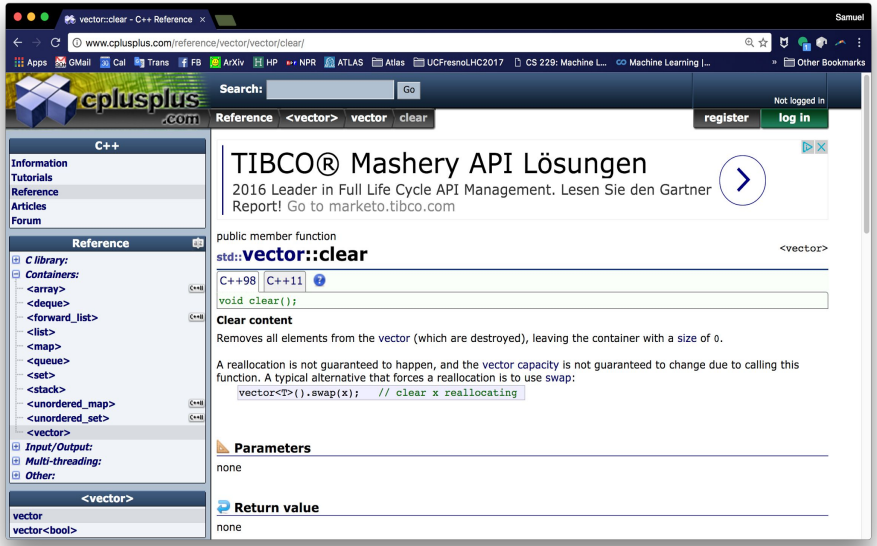
Self-educating : Code documentation

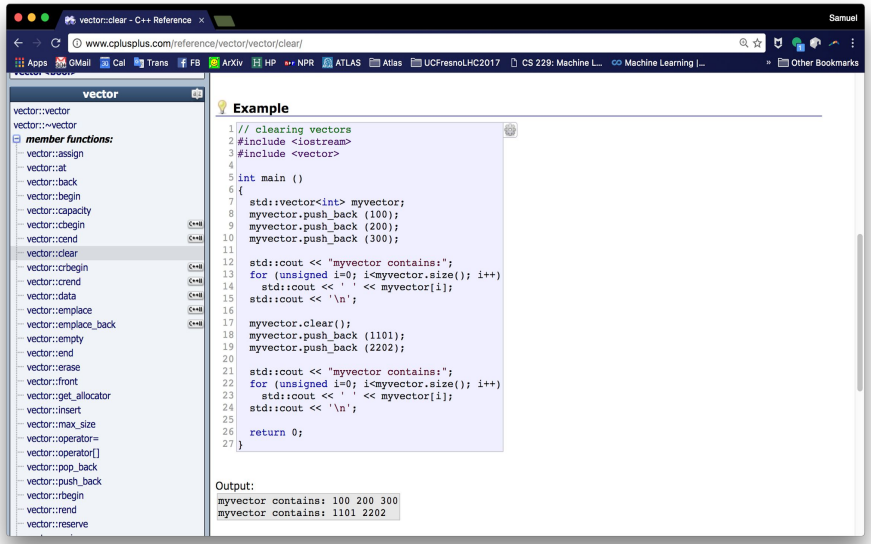
- Eventually, you see that there are descriptions of things you can do with a vector
  - **.at(#)** was used to access the # element of the vector
  - **.clear()** seems promising : most code is hotlinked to each other



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

- ASCII standards - http://www.cplusplus.com/doc/ascii/
  - 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