C++ CONTAINERS & SMART POINTERS

NOW WHERE DID I PUT THAT ITEM?

TYPES OF CONTAINERS

- Array
  - Items stored consecutively in RAM, cannot shrink/extend once made
- List
  - Unordered linked list of items scattered around RAM
- Vector
  - Items stored consecutively in RAM, can shrink/extend once made
- Map (Dictionary)
  - <key , item> pair stored as a binary tree for fast retrieval
- DeQueue
  - Double ended Queue, has head and tail insertion operators

More at http://www.cplusplus.com/reference/stl/

Also see: https://github.com/gibsjose/cpp-cheat-sheet/blob/master/Data%20Structures%20and%20Algorithms.md

EACH CONTAINER HAS BENEFITS & DRAWBACKS

- The choice of container will depend on

1. How you'd like to interact with the stored objects
    - Iteration
    - Storage / Retrieval
    - Performance vs. convenience
    - RAM use vs performance
2. What you are storing
    - Items of the same or different sizes
    - Simple or complex types (i.e. variables or whole classes)

There is a good chance you will start with one type of container and at some point change your mind, so items you store should be container "agnostic"

WHAT SHOULD YOU STORE IN A CONTAINER?

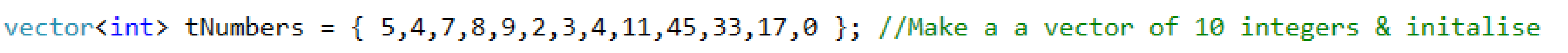
- Primitive types √
- Primitive types <int, float, string> √
- Pointers *tObject √ **(as long as you take care of delete)**
- SmartPointers unique_ptr<>, shared_ptr<> √
- Items on the stack? ✗ **NOOOOOO**
  - **They will contain garbage once function exits!**

TYPICAL CONTAINER ACTIONS

1. Add items
   - Some containers allow random insertion others just allow head/tail insertion
2. Remove Items
   - Some containers allow random removal others just allow head/tail removal
3. Iterate Items (step thought them)
   - All container support a form iteration
4. Find Item(s)
   - All container support a form find
5. Sort Items
   - Most containers allow a form of sorting

CREATING A CONTAINER

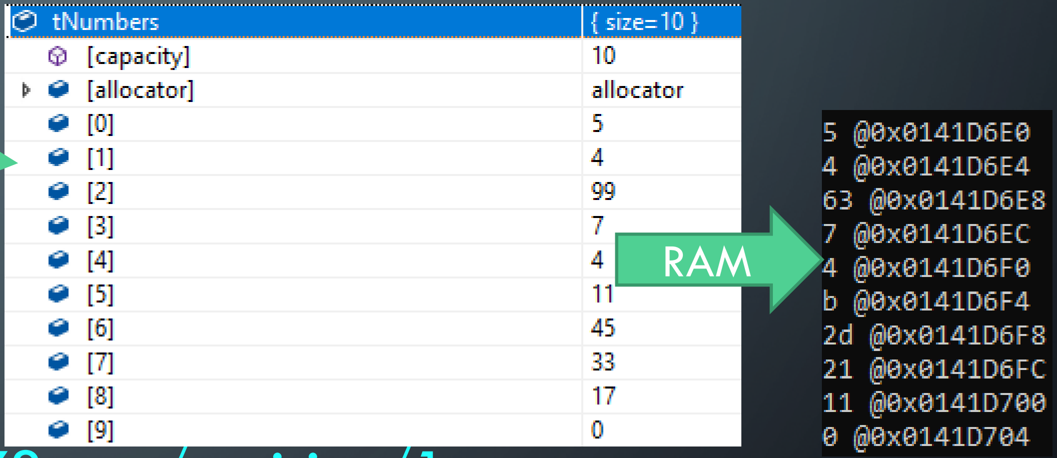
- Suppose we want to store 10 int's



```
vector<int> tNumbers = { 5,4,7,8,9,2,3,4,11,45,33,17,0 }; //Make a a vector of 10 integers & initalise
```

  - Quite similar to an array in C#

- This will create a new vector container and the values in {} will be placed within it

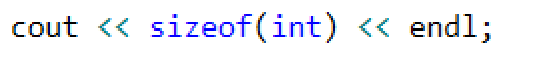


tNumbers	{ size=10 }
[capacity]	10
[allocator]	allocator
[0]	5
[1]	4
[2]	99
[3]	7
[4]	4
[5]	11
[6]	45
[7]	33
[8]	17
[9]	0

```
5 @0x0141D6E0
4 @0x0141D6E4
63 @0x0141D6E8
7 @0x0141D6EC
4 @0x0141D6F0
b @0x0141D6F4
2d @0x0141D6F8
21 @0x0141D6FC
11 @0x0141D700
0 @0x0141D704
```

- If not sure about hex (hexadecimal check) https://www.bbc.co.uk/bitesize/guides/zp73wmn/revision/1

- Note all the items are 4 bytes apart **sizeof**(int) is 4 bytes that's how much RAM it takes per item



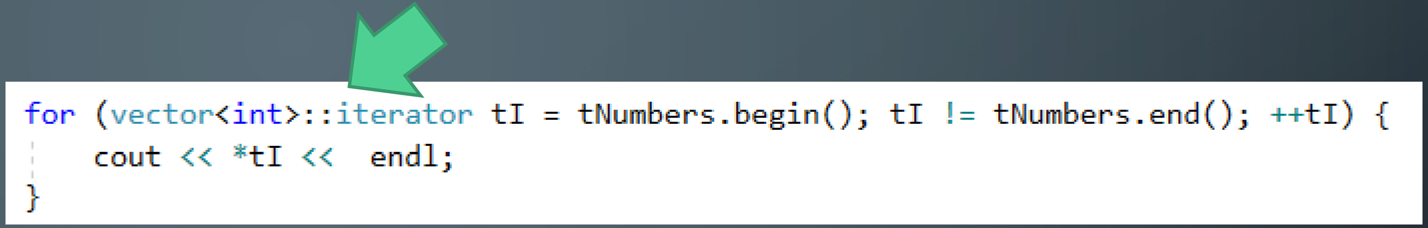
```
cout << sizeof(int) << endl;
```

ITERATION

- Traversing a collection, item by item
    - We have already done this with integers using a for(**iterator**, condition, next) loop
- The **iterator** is a special type of value which;
    1. Is a representation of the position within the collection of current value
    2. Allows us to access the current value via * the dereference operator
    3. Allows us to check if we are at the start or end of the collection
       .begin() & .end()
    4. Returns an iteration which represent the next or previous value ++ & --
- For the integers we have met in a for() the iterators are trivial
    - Numbers run in sequence, to the next number is just the current one +1, they can be their own iterators
- However inside a collection, the “iterator” is more complex

ITERATING THE vector<int>

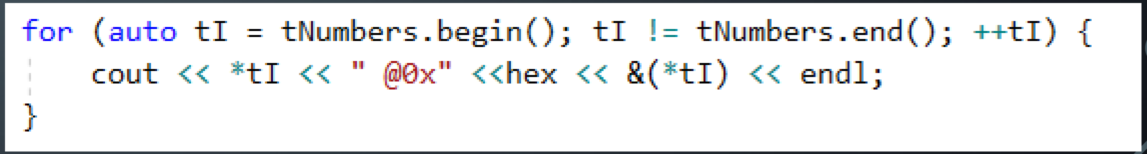
- The vector<> class has a definition for its own iterator



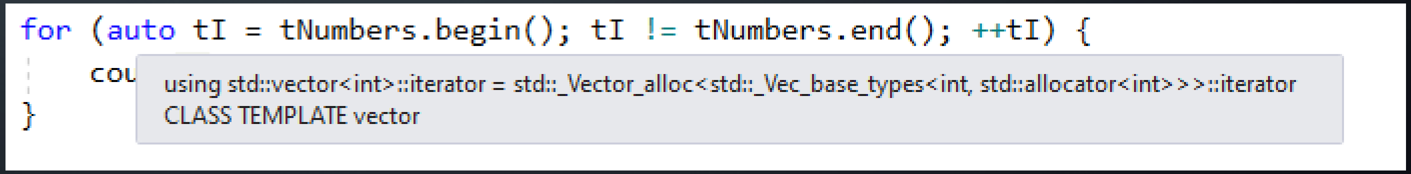
```
for (vector<int>::iterator tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
    cout << *tI <<  endl;
}
```

The syntax is pretty horrible, this is due to vector<> being a **template** class (more on these in a later lecture)

- However the new **auto** keyword can help
  - With auto the compiler will try to infer the type from its usage, and if it can work it out give you the correct type



```
for (auto tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
    cout << *tI << " @0x" <<hex << &(*tI) << endl;
}
```

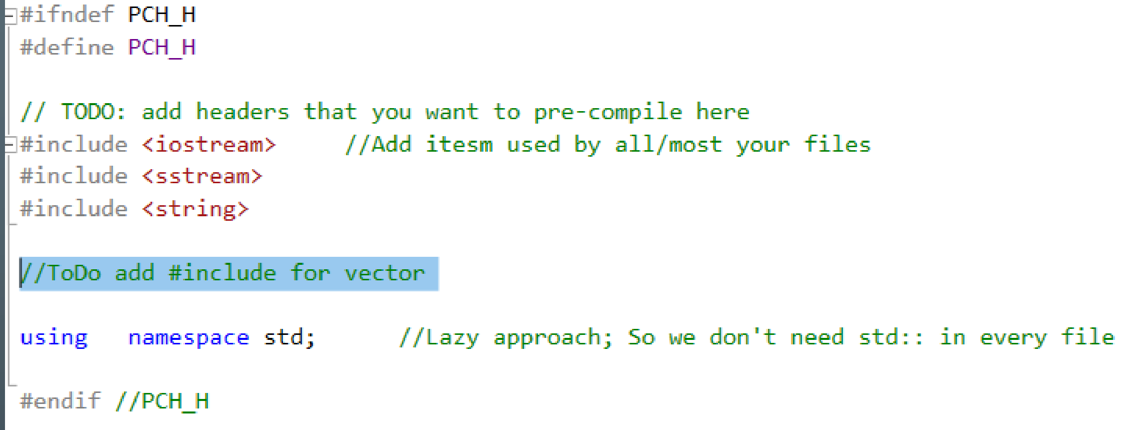


```
for (auto tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
    cou
}
```

using std::vector<int>::iterator = std::_Vector_alloc<std::_Vec_base_types<int, std::allocator<int>>>::iterator
CLASS TEMPLATE vector

CLASS TASK: MAKE A NEW PROJECT

1. C++, Console project called “Inventory”, follow instructions from last week
2. Compile it to make sure “Hello World” works
3. Inside pch.h
   - include the standard include files
   - Add the correct #include for vector as well



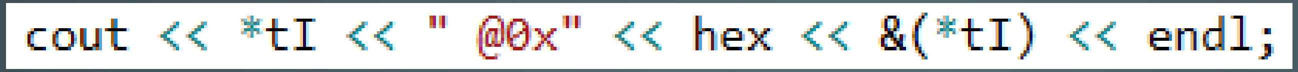
```
#ifndef PCH_H
#define PCH_H

// TODO: add headers that you want to pre-compile here
#include <iostream>     //Add itesm used by all/most your files
#include <sstream>
#include <string>

//ToDo add #include for vector

using   namespace std;      //Lazy approach; So we don't need std:: in every file

#endif //PCH_H
```

4. Test again
5. Inside main()
   - Remove “hello world” code
   - Add code to make a vector<int> initialised with 10 numbers of your choosing (see pervious slides)
   - Add code to print the numbers stored in vector

- Test, make sure your code compiles and shows your 10 numbers
  NB: you can use an explicitly defined Iterator or use the **<u>auto</u>** keyword

HOUSEKEEPING ITEMS (GOOD TO KNOW STUFF)

- The & means you want to deal with the address of the item, rather than its value



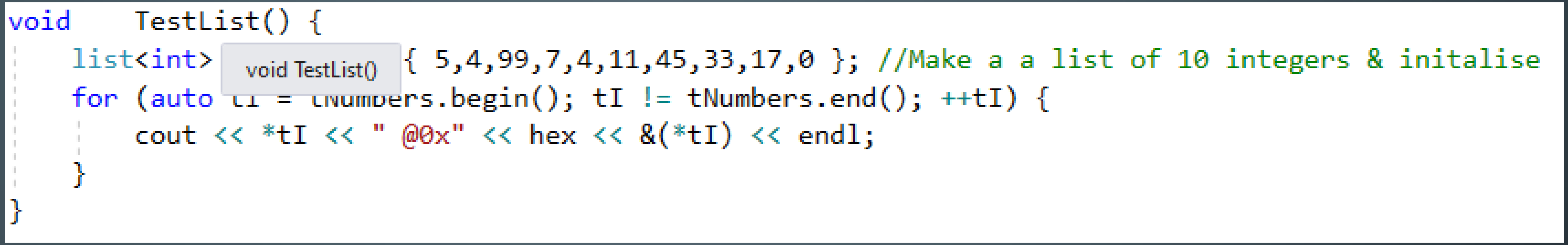
```
cout << *tI << " @0x" << hex << &(*tI) << endl;
```

- * deferences the iterator, then & takes its memory location, use brackets to make your intentions on priority clear, i.e. dereference before taking address
- << **hex** << tells the output stream to print hex values rather than decimal
- Role of **begin()**, **end()** and why we use **++tIterator**, and not tIterator++
  - begin() is the item BEFORE the first one
  - end() is the item AFTER the last one

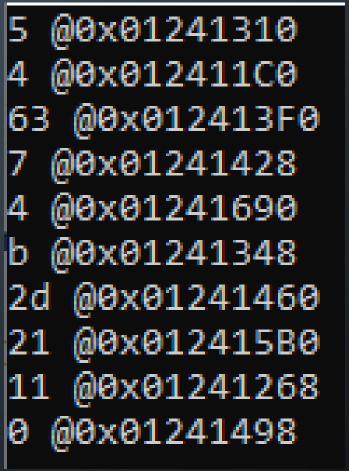
  So to auto tI = tNumbers.begin() actually references the item before the first one,
  ++tIterator will pre-increment to go to actual first item
  Also tI != tNumbers.end() checks for the item after the last one to end the run

list<int>

- An unordered collection, which can grow and shrink



```
void    TestList() {
    list<int>                { 5,4,99,7,4,11,45,33,17,0 }; //Make a a list of 10 integers & initalise
    for (auto tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
        cout << *tI << " @0x" << hex << &(*tI) << endl;
    }
}
```



```
5 @0x01241310
4 @0x012411C0
63 @0x012413F0
7 @0x01241428
4 @0x01241690
b @0x01241348
2d @0x01241460
21 @0x012415B0
11 @0x01241268
0 @0x01241498
```

- However unlike vector<int> they are not stored consecutively in RAM. The next item could be anywhere in RAM
- list<> maintains order by using links to the previous and next item internally

CLASS TASK

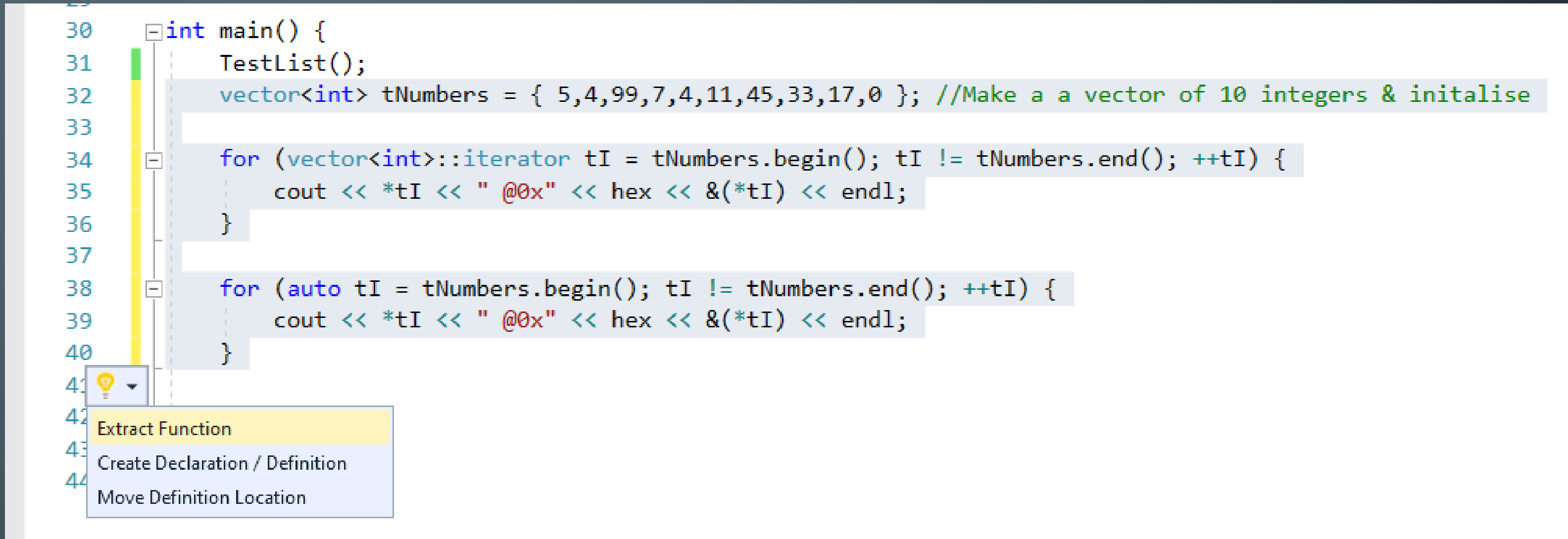
- Refactor you code to move your vector<int> test code into its own function

  Select code & right click

  QuickAction

  Extract function

  Name it TestVector



```
int main() {
    TestList();
    vector<int> tNumbers = { 5,4,99,7,4,11,45,33,17,0 }; //Make a a vector of 10 integers & initalise

    for (vector<int>::iterator tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
        cout << *tI << " @0x" << hex << &(*tI) << endl;
    }

    for (auto tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
        cout << *tI << " @0x" << hex << &(*tI) << endl;
    }
```

- If that does not work

  Cut & Paste

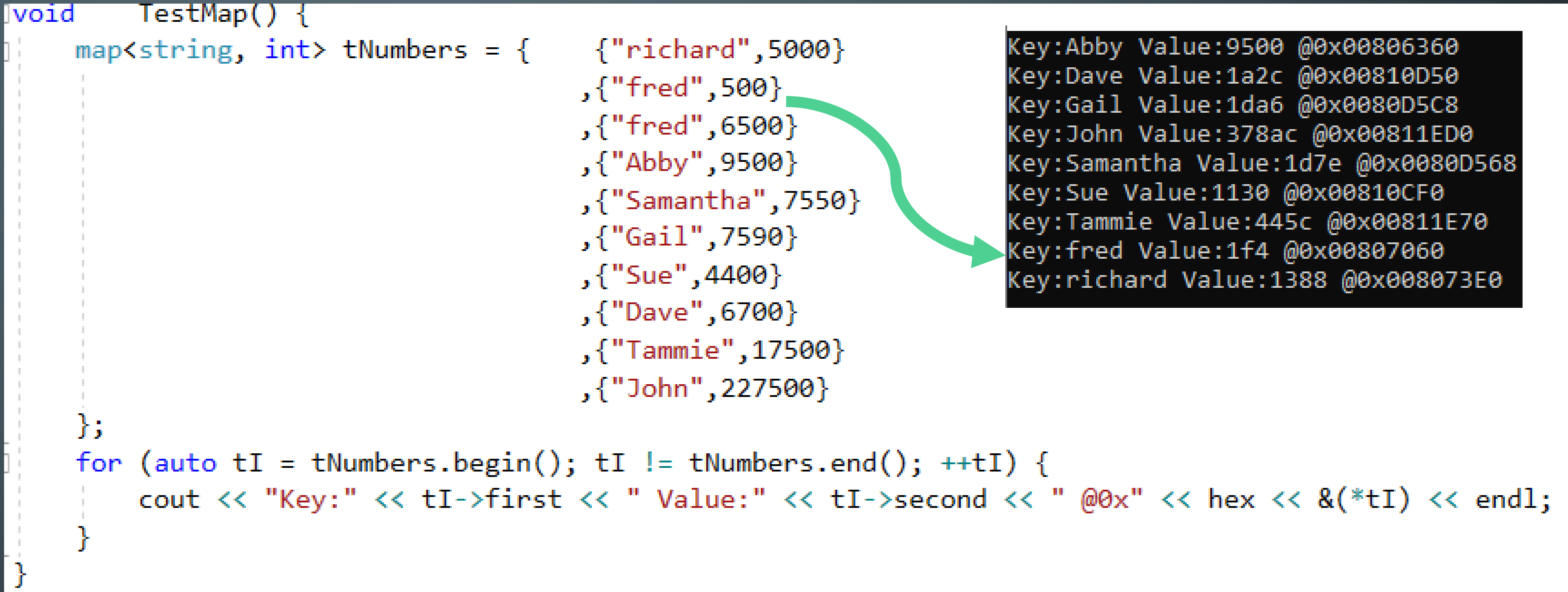
- Make another function called TestList and add code to do the same as TestVector, but using a list

  NB: you need to #include <list> in pch.h for to get access to list<>

- Test your code

The map<key , item>

- A map is also known as a dictionary
- It allows item lookup by key
- The iterator will return first == key second == value
- Maps cannot have duplicate keys note



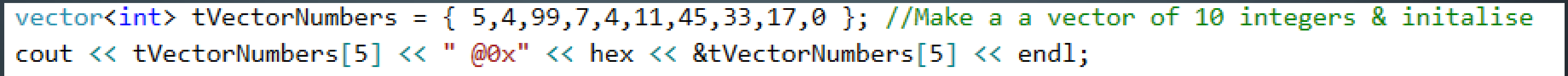
```
void    TestMap() {
    map<string, int> tNumbers = {    {"richard",5000}
                                    ,{"fred",500}
                                    ,{"fred",6500}
                                    ,{"Abby",9500}
                                    ,{"Samantha",7550}
                                    ,{"Gail",7590}
                                    ,{"Sue",4400}
                                    ,{"Dave",6700}
                                    ,{"Tammie",17500}
                                    ,{"John",227500}
    };
    for (auto tI = tNumbers.begin(); tI != tNumbers.end(); ++tI) {
        cout << "Key:" << tI->first << " Value:" << tI->second << " @0x" << hex << &(*tI) << endl;
    }
}
```

```
Key:Abby Value:9500 @0x00806360
Key:Dave Value:1a2c @0x00810D50
Key:Gail Value:1da6 @0x0080D5C8
Key:John Value:378ac @0x00811ED0
Key:Samantha Value:1d7e @0x0080D568
Key:Sue Value:1130 @0x00810CF0
Key:Tammie Value:445c @0x00811E70
Key:fred Value:1f4 @0x00807060
Key:richard Value:1388 @0x008073E0
```

when fred is included twice above the 2nd inclusion overwrites the first

Choosing the correct container

- Will the number of items change frequently?
  - Each time a vector grows beyond its default allocation a new larger one is created and the old items are copied to it (slow)
  - Lists just grow one item at a time
- Do you typically need to access items in sequence or randomly
  - vectors are fast for direct access and can be accessed via



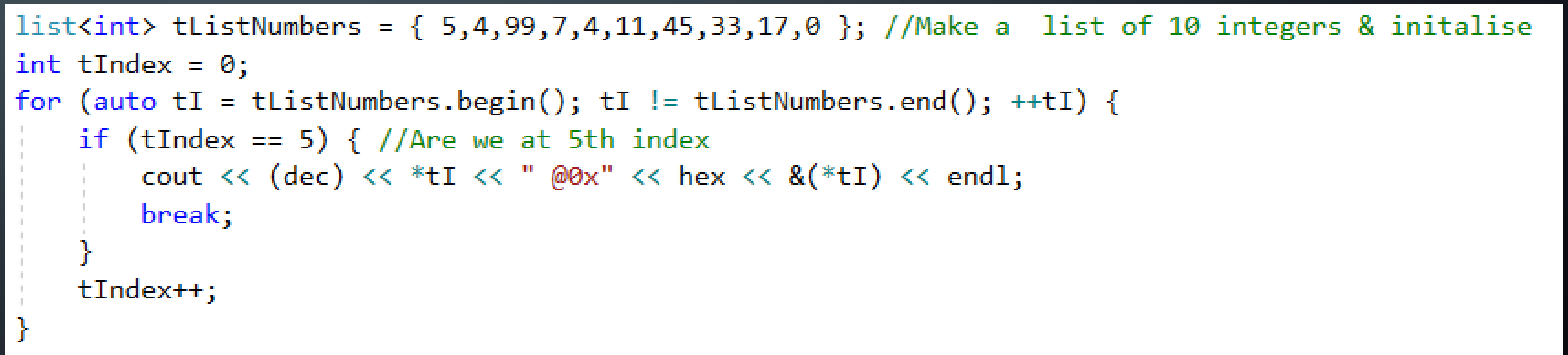
```
vector<int> tVectorNumbers = { 5,4,99,7,4,11,45,33,17,0 }; //Make a a vector of 10 integers & initalise
cout << tVectorNumbers[5] << " @0x" << hex << &tVectorNumbers[5] << endl;
```

  - lists are fast for next item access, but getting a specific item means you need to search the whole list till you find it

    NB:#include <algorithm>

    contains std::find()

    which does this



```
list<int> tListNumbers = { 5,4,99,7,4,11,45,33,17,0 }; //Make a  list of 10 integers & initalise
int tIndex = 0;
for (auto tI = tListNumbers.begin(); tI != tListNumbers.end(); ++tI) {
    if (tIndex == 5) { //Are we at 5th index
        cout << (dec) << *tI << " @0x" << hex << &(*tI) << endl;
        break;
    }
    tIndex++;
}
```

SORTING A BUNCH OF NUMBERS

- Bubblesort
  - Standard algorithm where the values “bubble to the top”
- Iterates container from start to finish
  - If current item is > (greater than) than next item, swap them
- We know we have done them all if we compete a whole pass without any swapping
- Gotcha’s
  - As we need to access both the current and the next value we must make sure we don’t step outside array bounds, containers include a getter size() which gets an item count
  - When we swap items we must not accidently overwrite them, we can use a temp variable for this

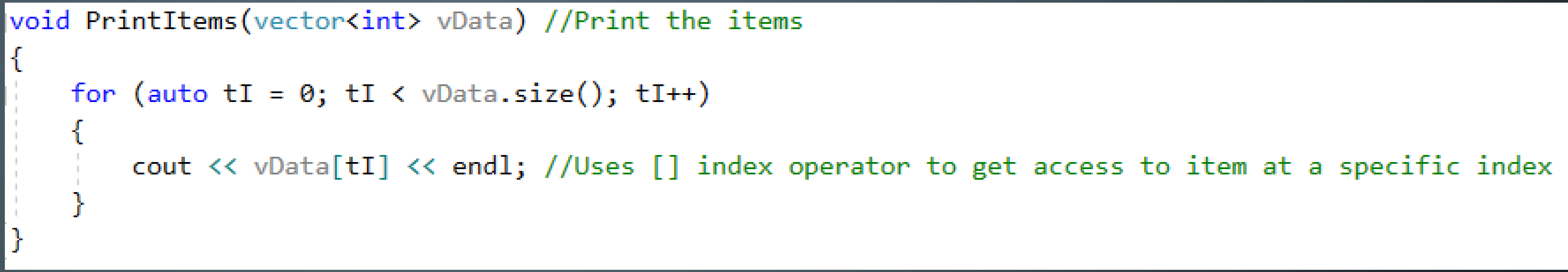
CLASS TASK

- Create a PrintItems() function which will print your vector<int>

Note the use of []

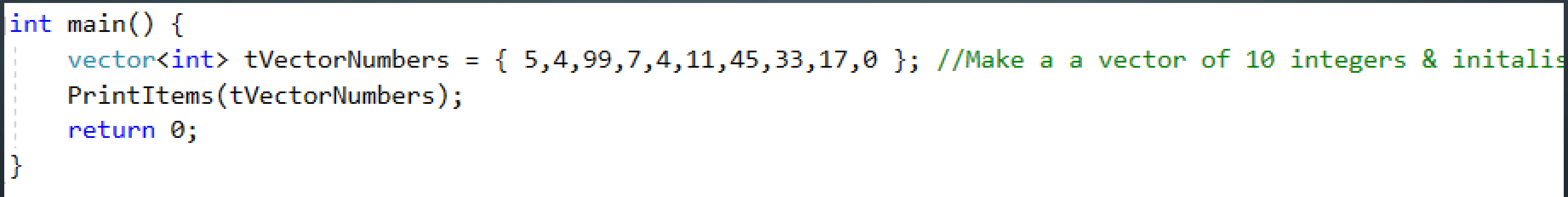
to access the item

by its index



```
void PrintItems(vector<int> vData) //Print the items
{
    for (auto tI = 0; tI < vData.size(); tI++)
    {
        cout << vData[tI] << endl; //Uses [] index operator to get access to item at a specific index
    }
}
```

- Modify your main to test to see if you can print the set of numbers you made



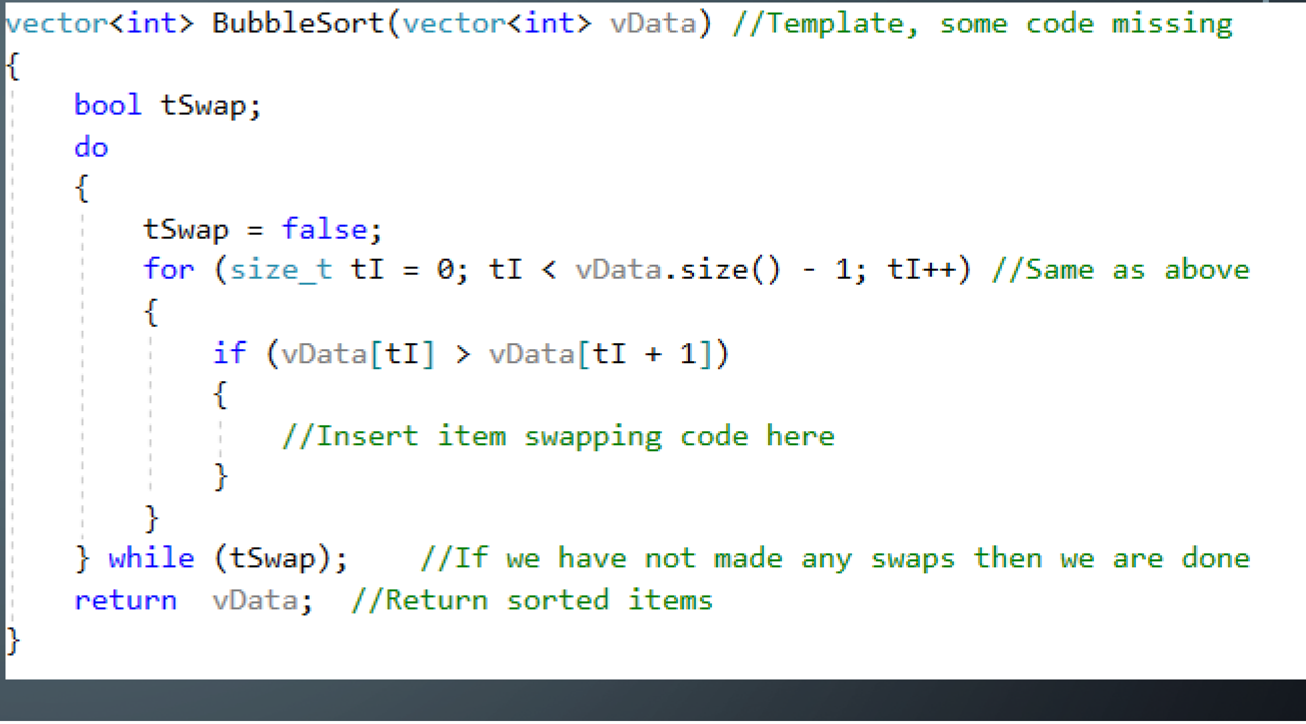
```
int main() {
    vector<int> tVectorNumbers = { 5,4,99,7,4,11,45,33,17,0 }; //Make a a vector of 10 integers & initalis
    PrintItems(tVectorNumbers);
    return 0;
}
```

- Now make a copy of the function and call it BubbleSort() & try to implement the sort algorithm, then test

NB: Hints on next slide, but have go before peeking

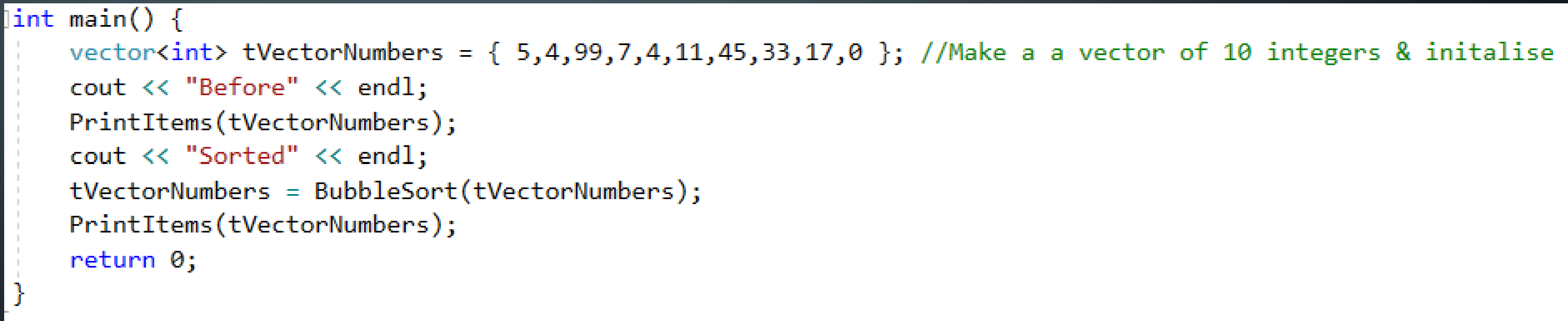
HINT: FUNCTION OUTLINE

- You need to add code to do the swap into the template



```
vector<int> BubbleSort(vector<int> vData) //Template, some code missing
{
    bool tSwap;
    do
    {
        tSwap = false;
        for (size_t tI = 0; tI < vData.size() - 1; tI++) //Same as above
        {
            if (vData[tI] > vData[tI + 1])
            {
                //Insert item swapping code here
            }
        }
    } while (tSwap);    //If we have not made any swaps then we are done
    return  vData;  //Return sorted items
}
```

- Test



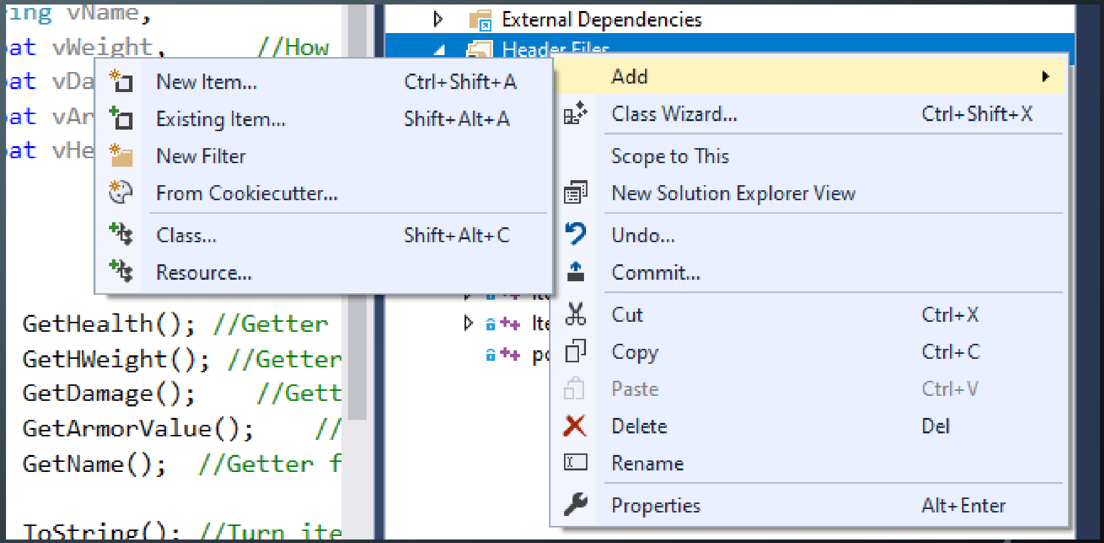
```
int main() {
    vector<int> tVectorNumbers = { 5,4,99,7,4,11,45,33,17,0 }; //Make a a vector of 10 integers & initalise
    cout << "Before" << endl;
    PrintItems(tVectorNumbers);
    cout << "Sorted" << endl;
    tVectorNumbers = BubbleSort(tVectorNumbers);
    PrintItems(tVectorNumbers);
    return 0;
}
```

STORING COMPLEX TYPES

- Pointers vs smart pointers
  - Both are refences to (point to) objects
  - However, smart pointers will automatically destroy/delete the object they reference when they go out of scope
- 2 types
  - unique_ptr<> //a single reference to an item, cannot be copied or shared
  - shared_ptr<> //multiple references to an item, can be shared & copied, will delete item when no more references are in scope, it’s the most useful one to use
  - Both live in std:: namespace
- Perfect for storing in containers

USING LAST WEEKS ITEM CLASS

- Copy the Item.cpp & Item.h files from last weeks project, make sure you **<u>COPY</u>** not move to the current one to the same place where your current .cpp &.h files are
- Use add existing item to add to this project add the header to headers and the source to source

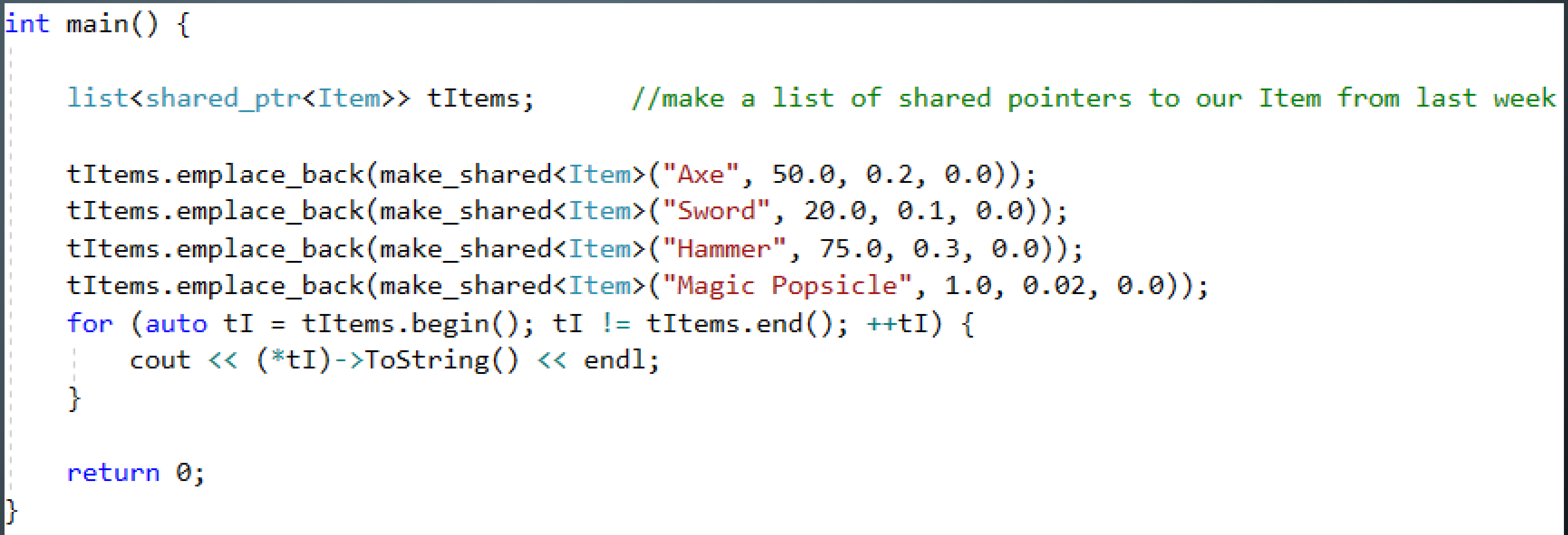


```
ing vName,
at vWeight,        //How
at vDa
at vAr
at vHe

 GetHealth(); //Getter
 GetHWeight(); //Getter
 GetDamage();     //Gett
 GetArmorValue();     //
 GetName();  //Getter f

 ToString(); //Turn ite
```

- Add #include “Item.h” below #include “pch.h” in the file which contains main()
- Make sure it still compiles

TASK: AMEND MAIN()

- If you want to keep current code comment it out or refactor to a function (see slides)
- Add this code
- Compile & run

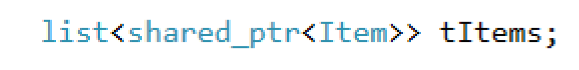


```
int main() {

    list<shared_ptr<Item>> tItems;      //make a list of shared pointers to our Item from last week

    tItems.emplace_back(make_shared<Item>("Axe", 50.0, 0.2, 0.0));
    tItems.emplace_back(make_shared<Item>("Sword", 20.0, 0.1, 0.0));
    tItems.emplace_back(make_shared<Item>("Hammer", 75.0, 0.3, 0.0));
    tItems.emplace_back(make_shared<Item>("Magic Popsicle", 1.0, 0.02, 0.0));
    for (auto tI = tItems.begin(); tI != tItems.end(); ++tI) {
        cout << (*tI)->ToString() << endl;
    }

    return 0;
}
```

- You are now making your items as smart pointers, note how they automatically delete themselves

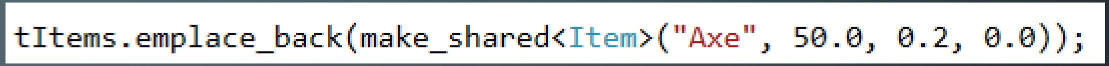
BREAKDOWN

- The list will be a list of shared_ptr<Item> of type Item



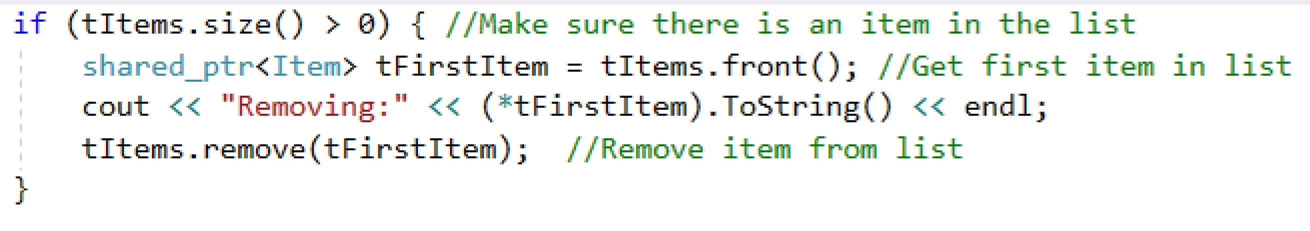
```
list<shared_ptr<Item>> tItems;
```

- make_shared<Item>(your values for the item) allocated RAM and makes the shared_prt



```
tItems.emplace_back(make_shared<Item>("Axe", 50.0, 0.2, 0.0));
```

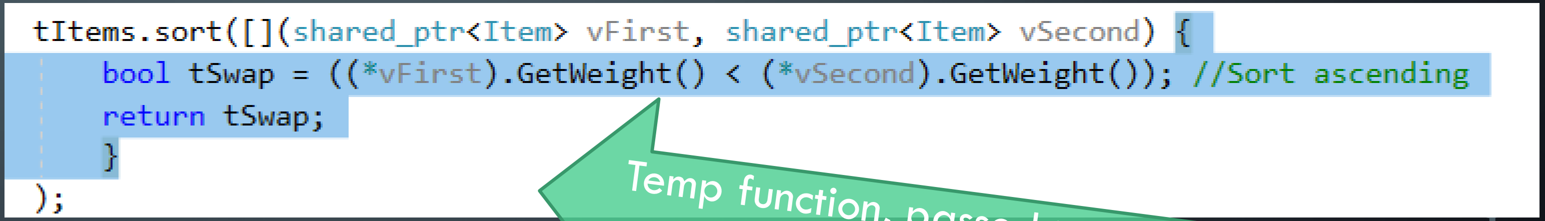
- items.emplace_back() adds the item to the list
  - You can also use push_back() however emplace avoids an extra copy operation as it makes the item in place vs. making it and then coping it.
- Lists can have items removed, and with smart pointers when the reference goes, the item is deleted



```
if (tItems.size() > 0) { //Make sure there is an item in the list
    shared_ptr<Item> tFirstItem = tItems.front(); //Get first item in list
    cout << "Removing:" << (*tFirstItem).ToString() << endl;
    tItems.remove(tFirstItem);  //Remove item from list
}
```

SORTING THE LIST USING std::sort

- Say for example we want to sort the items in our list by weight we can use
  - std::sort lives in #include <algorithm>, this needs to be in pch.h of it to work
  - ➢It can sort items in containers (more effectively then BubbleSort)
  - ➢We know our item has weight, but C++ list<Item> has no idea how we want to sort
- Enter the **Lambda** function
  - A temporary function which can be passed to std::sort(), to tell it how we want to sort (or more specifically if items should be swapped)
  - The syntax is a bit horrible basically you are passing a temp function telling sort how to sort, into sort and its called for every item



```
tItems.sort([](shared_ptr<Item> vFirst, shared_ptr<Item> vSecond) {
    bool tSwap = ((*vFirst).GetWeight() < (*vSecond).GetWeight()); //Sort ascending
    return tSwap;
    }
);
```

Temp function, passed to sort, returns true if swap needed

  - This is a lambda function [capture](variables) {definition} capture allow you to access calling scope variables either by ref [&] or by copy [=], we don't need any captures

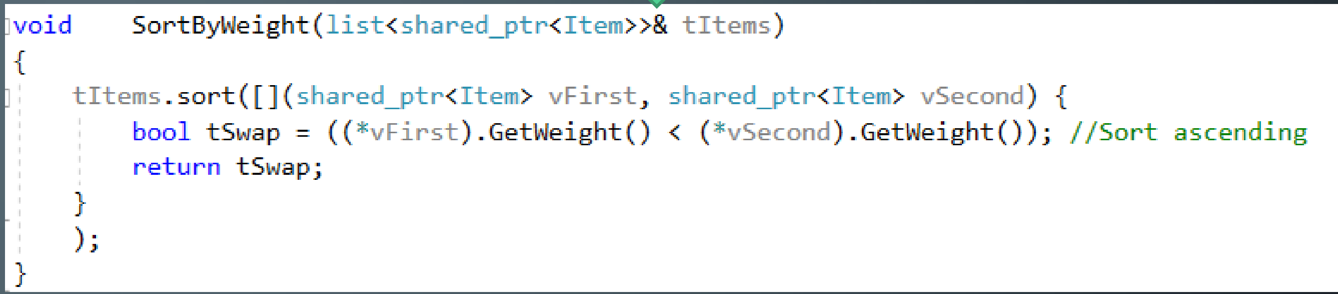
TASK: ADD A SORT BY DAMAGE, HIGHEST FIRST

1. Refactor your code to move the current sort into its own function, called SortByWeight()

   You cannot use quickaction for this, so its cut & paste

   Also you need to pass in the list to sort by ref (&) not value as your function is changing it

ref



```
void    SortByWeight(list<shared_ptr<Item>>& tItems)
{
    tItems.sort([](shared_ptr<Item> vFirst, shared_ptr<Item> vSecond) {
        bool tSwap = ((*vFirst).GetWeight() < (*vSecond).GetWeight()); //Sort ascending
        return tSwap;
    }
    );
}
```

2. Make a copy called SortByDamage() and amend it to sort items in descending damage order by changing the lambda function
3. Test

Q&A

+ CATCHUP, SO JUST ASK ME IF YOU ARE STUCK ON ANY OF THE MATERIAL