MNXB01 course - C++ module

Caterina Marcon

caterina.marcon@hep.lu.se

Lecture's goals

You will learn:

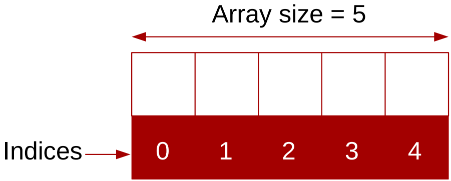
- how to use arrays, vectors and strings;
- how to use lists, pairs and sets;
- to declare pointers.

Containers: arrays

- A container is a holder object that stores a collection of objects (its elements). There are several types of containers including arrays, vectors and strings;
- Arrays are fixed-size sequence containers: they hold a specific number of elements ordered in a strict linear sequence;
- An array must be declared before it is used:

[C++] Array declaration

```cpp
type name [number of elements];
int arr [5];
```



Containers: arrays

[C++] Compute the sum of the elements of the array

```cpp
#include <iostream> //for cout and cin
using namespace std;

int main() {
 const int length = 10; //constant variable indicating the size of the array
 int a[length] = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};

 //Sum contents of array a
 for (int i = 0; i < arraySize; i++) {
   total = total + a[i];
   }
 cout << "Total of array elements: "<< total << endl;

    return 0;
}
```

- constant variables must be initialized with a constant expression when they are declared and cannot be modified thereafter.

Containers: arrays

[C++]

```cpp
#include <iostream> //for cout and cin
using namespace std;

int main() {
 const int length = 10; //the length must be known at the compile time
 int arr[length]; //the array is fixed-size
 int input;
 int pos = 0; //an array doesn't know its own size or how many elements it contains

 cout << "Enter an integer number: "<< endl;
 //This loop means "keep reading values from cin into x, and continue looping".
 while(cin >> input) {
    arr[pos] = input;
    if (pos == length) break; //Remember that the array can't grow, this is a limit
    ++pos; //we have to keep track of the position
 }
 for (int i=0; i < pos; i++) cout << arr[i] << endl;
    return 0;
}
```

Containers: arrays

You should avoid to use arrays:

- The length must be known at the compile time.
- The array is fixed-size.
- An array doesn't know its own size or how many elements it contains (we have to keep track of the position).
- Remember that the array can't grow: this is a limit.

Containers: vectors

- C++ class: class can be thought as a collection of data and functions. The member of a class are accessed with a . (dot). C++ provides different classes; vector is an example.
- C++ standard library class vector represents a more robust type of array featuring many additional capabilities.
- A vector is a sequential container that can change size dynamically.
- When the vector's memory is exhausted, the vector allocates a larger contiguous area of memory, copies the original elements into the new memory and de-allocates the old memory.

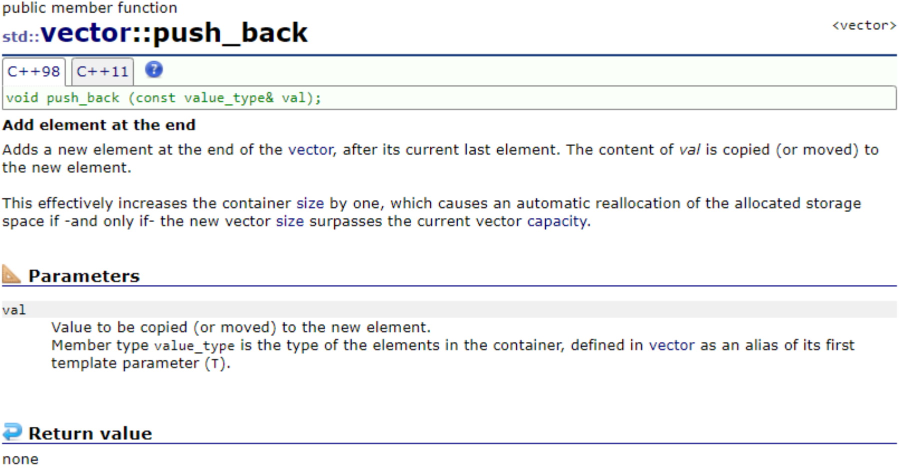
Containers: vectors

[C++] How to create a vector, store each input and print them back

```cpp
#include <iostream> //For cout and cin
#include <vector> //For vectors
using namespace std;

int main() {
    vector<int> vec; //create a vector with base type int
    int input;
    cout << "Enter an integer number: "<< endl;
    while (cin >> input) vec.push_back(input); //store each input
    //print them back
    for (size_t i = 0; i < vec.size(); ++i) cout << vec.at(i) << endl;
    return 0;
}
```

Containers: vectors



public member function

std::vector::push_back

<vector>

```
void push_back (const value_type& val);
```

Add element at the end

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Parameters

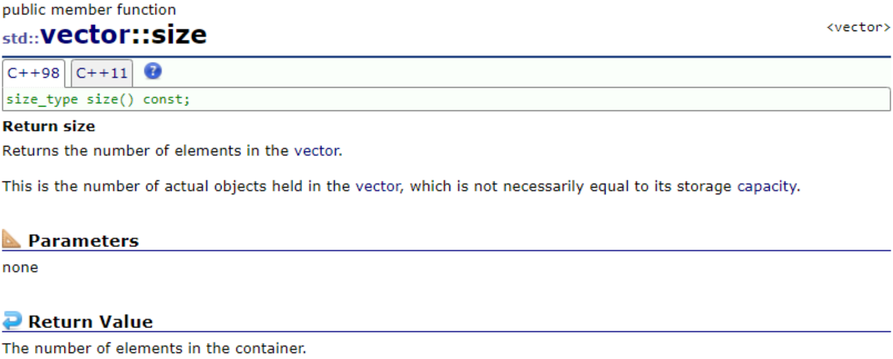
val

Value to be copied (or moved) to the new element.
Member type `value_type` is the type of the elements in the container, defined in vector as an alias of its first template parameter (`T`).

Return value

none

Containers: vectors



public member function

std::vector::size

```
size_type size() const;
```

Return size

Returns the number of elements in the vector.

This is the number of actual objects held in the vector, which is not necessarily equal to its storage capacity.

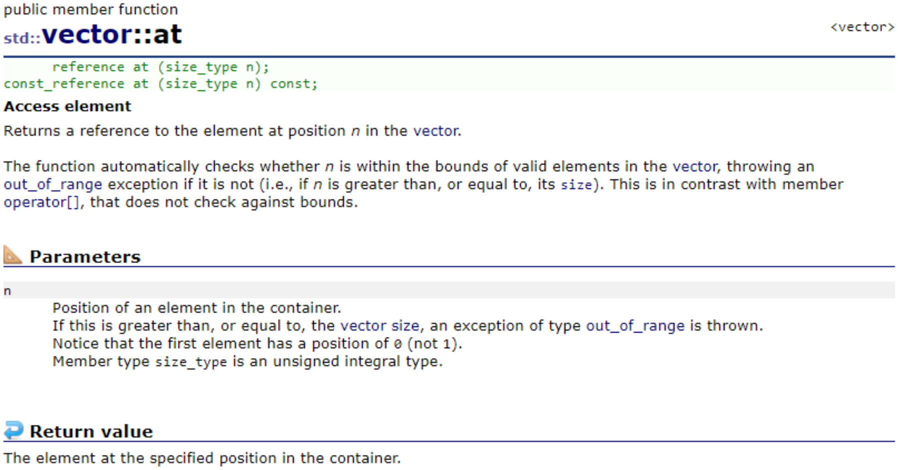
Parameters

none

Return Value

The number of elements in the container.

Containers: vectors



public member function

std::vector::at

<vector>

```
      reference at (size_type n);
const_reference at (size_type n) const;
```

Access element

Returns a reference to the element at position *n* in the vector.

The function automatically checks whether *n* is within the bounds of valid elements in the vector, throwing an out_of_range exception if it is not (i.e., if *n* is greater than, or equal to, its `size`). This is in contrast with member operator[], that does not check against bounds.

Parameters

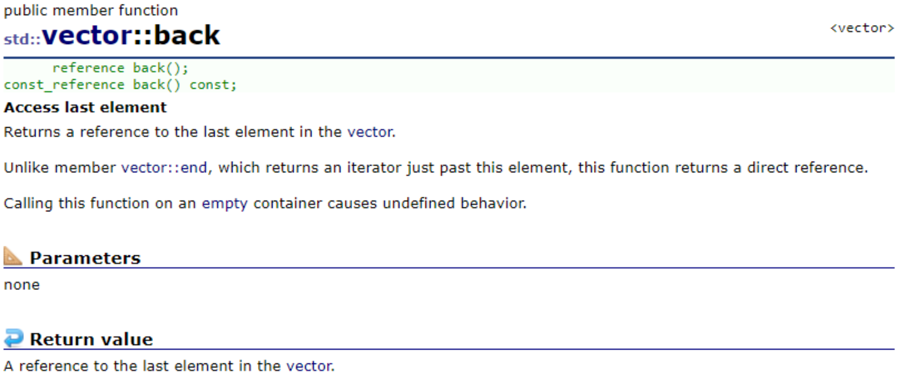
n

Position of an element in the container.
If this is greater than, or equal to, the vector size, an exception of type out_of_range is thrown.
Notice that the first element has a position of `0` (not `1`).
Member type `size_type` is an unsigned integral type.

Return value

The element at the specified position in the container.

Containers: vectors



public member function

std::vector::back

<vector>

```
      reference back();
const_reference back() const;
```

Access last element

Returns a reference to the last element in the vector.

Unlike member vector::end, which returns an iterator just past this element, this function returns a direct reference.

Calling this function on an empty container causes undefined behavior.

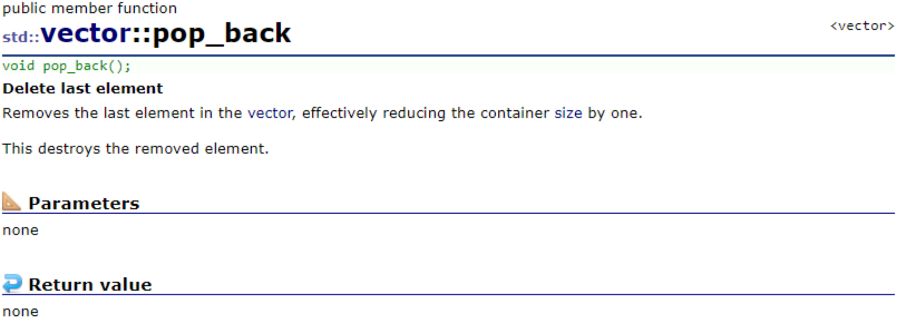
Parameters

none

Return value

A reference to the last element in the vector.

Containers: vectors



public member function

std::vector::pop_back

<vector>

```
void pop_back();
```

Delete last element

Removes the last element in the vector, effectively reducing the container size by one.

This destroys the removed element.

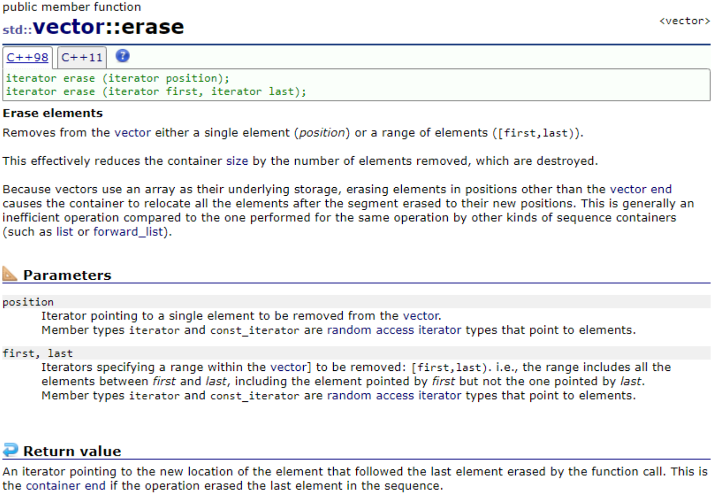
Parameters

none

Return value

none

Containers: vectors



public member function

std::vector::erase

<vector>

C++98 | C++11 | ?

```
iterator erase (iterator position);
iterator erase (iterator first, iterator last);
```

Erase elements

Removes from the vector either a single element (*position*) or a range of elements ([first,last)).

This effectively reduces the container size by the number of elements removed, which are destroyed.

Because vectors use an array as their underlying storage, erasing elements in positions other than the vector end causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

Parameters

position

Iterator pointing to a single element to be removed from the vector.
Member types iterator and const_iterator are random access iterator types that point to elements.

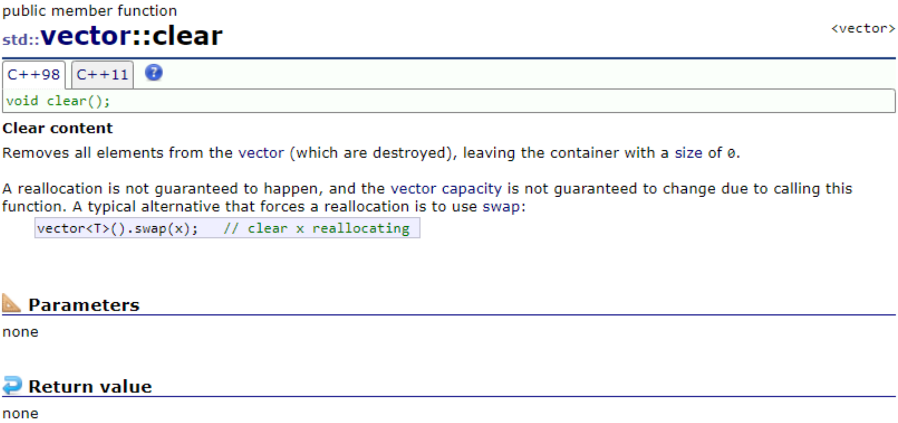
first, last

Iterators specifying a range within the vector] to be removed: [first,last). i.e., the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.
Member types iterator and const_iterator are random access iterator types that point to elements.

Return value

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

Containers: vectors



public member function

std::vector::clear

```
void clear();
```

Clear content

Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

A reallocation is not guaranteed to happen, and the vector capacity is not guaranteed to change due to calling this function. A typical alternative that forces a reallocation is to use swap:

```
vector<T>().swap(x);   // clear x reallocating
```

Parameters

none

Return value

none

Containers: strings

- A string is a sequence of characters, implemented by the string class;

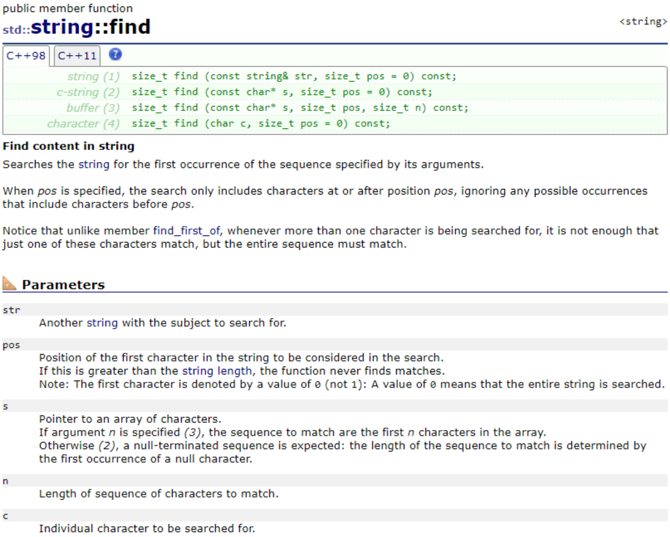
[C++]

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

int main(){
    string str ("It is dangerous to go alone, take this!");
    size_t pos = str.find("take"); //position in string where "take" is found

    cout << str.substr(0, 18) << str.substr(pos) << endl;
    return 0;
}
```

Containers: strings



public member function

std::string::find

<string>

C++98 C++11 ?

```
   string (1)  size_t find (const string& str, size_t pos = 0) const;
 c-string (2)  size_t find (const char* s, size_t pos = 0) const;
   buffer (3)  size_t find (const char* s, size_t pos, size_t n) const;
character (4)  size_t find (char c, size_t pos = 0) const;
```

Find content in string

Searches the string for the first occurrence of the sequence specified by its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences that include characters before *pos*.

Notice that unlike member find_first_of, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.

Parameters

`str`

Another string with the subject to search for.

`pos`

Position of the first character in the string to be considered in the search.
If this is greater than the string length, the function never finds matches.
Note: The first character is denoted by a value of `0` (not `1`): A value of `0` means that the entire string is searched.

`s`

Pointer to an array of characters.
If argument *n* is specified *(3)*, the sequence to match are the first *n* characters in the array.
Otherwise *(2)*, a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.

`n`

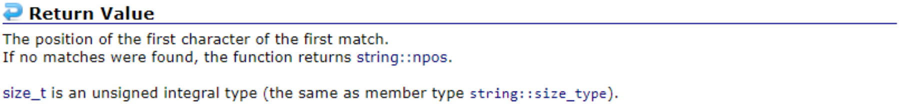
Length of sequence of characters to match.

`c`

Individual character to be searched for.

Containers: strings

More on `find()`



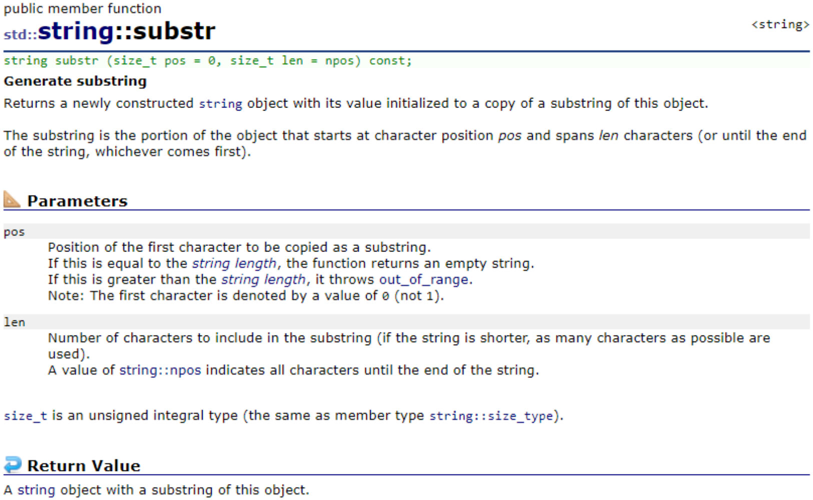
Return Value

The position of the first character of the first match.
If no matches were found, the function returns string::npos.

size_t is an unsigned integral type (the same as member type `string::size_type`).

Note that `string::npos` is just a constant integer. Its value is `-1` .

Containers: strings



public member function

std::string::substr

<string>

```
string substr (size_t pos = 0, size_t len = npos) const;
```

Generate substring

Returns a newly constructed `string` object with its value initialized to a copy of a substring of this object.

The substring is the portion of the object that starts at character position *pos* and spans *len* characters (or until the end of the string, whichever comes first).

Parameters

`pos`

Position of the first character to be copied as a substring.
If this is equal to the *string length*, the function returns an empty string.
If this is greater than the *string length*, it throws out_of_range.
Note: The first character is denoted by a value of `0` (not `1`).

`len`

Number of characters to include in the substring (if the string is shorter, as many characters as possible are used).
A value of string::npos indicates all characters until the end of the string.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

Return Value

A string object with a substring of this object.

Reading out line by line using strings

- For reading out lines of data from a file, you can use getline function.

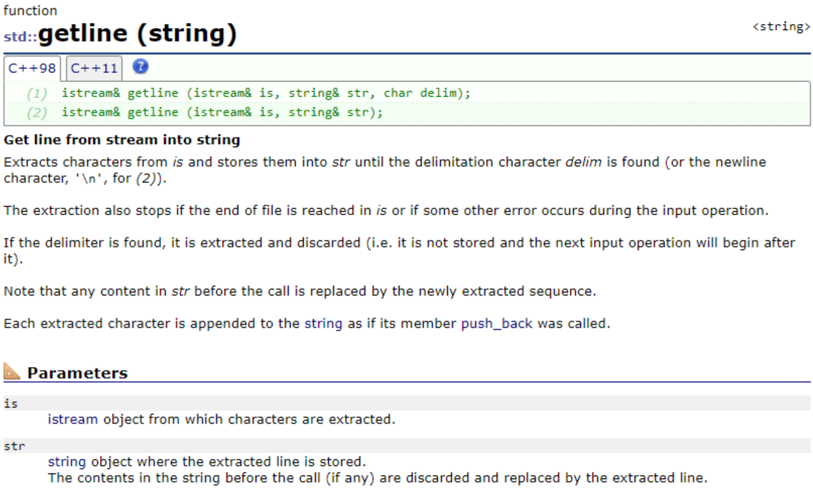
[C++]

```cpp
#include <iostream>
#include <fstream> //header file for file writing and reading
#include <string> //header file for strings
using namespace std;

int main(){
    fstream in;
    in.open("inputtext.txt"); //open an input file
    string s;
    cout<< "Line 1:" <<endl;
    getline(in,s); //read first line
    cout<<s<<endl;
    cout<< "Line 2:" <<endl;
    getline(in,s); //read second line
    cout<<s<<endl;

    in.close();
    return 0;
}
```

Containers: strings



function

std::getline (string)

<string>

```
(1) istream& getline (istream& is, string& str, char delim);
(2) istream& getline (istream& is, string& str);
```

Get line from stream into string

Extracts characters from *is* and stores them into *str* until the delimitation character *delim* is found (or the newline character, '\n', for *(2)*).

The extraction also stops if the end of file is reached in *is* or if some other error occurs during the input operation.

If the delimiter is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

Note that any content in *str* before the call is replaced by the newly extracted sequence.

Each extracted character is appended to the string as if its member push_back was called.

Parameters

is
: istream object from which characters are extracted.

str
: string object where the extracted line is stored.
The contents in the string before the call (if any) are discarded and replaced by the extracted line.

Lists

- A list is a container with fast element insertion and removal;
- unlike vectors, elements in a list have no absolute position. Use an iterator to loop through them. Iterators act similarly to pointers.

[C++] Lists

```cpp
#include <iostream> //for cout and cin
#include <list> //for list
using namespace std;

int main(){
 list <int> lst;//list with basetype int
 lst.push_back(10);//insert some elements, iterate over the list and print them
 lst.push_back(15);
 for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
  {
  cout << *it << endl;
  }
 return 0;
}
```

Containers: iterators

An `iterator` is an object that points to an element inside the container. We can use iterators to move through the contents of the container. They point to some location of the list and we can access content at that particular location using them.

[C++] iterators

```cpp
list<int> lst; // This is the actual list
list<int>::iterator it; // This is the iterator attached to it

it = lst.begin(); // The list knows where it begins
 it++; // The iterator can be moved forward to point to the next element
 it--; // or backwards

(*it); // this is the current element in the list to which
      // the iterator is pointing
it = lst.end(); // The list also knows where it ends
```

The same syntax is also valid for vectors and sets.

Pairs

- A pair is a simple container that stores two values.

[C++] Pairs

```
#include <iostream> //for cout and cin
#include <utility> //for pair
using namespace std;

int main(){
 pair <int, double> p(5, 3.14); //A pair of int and double
 cout << "The pair is " << p.first ", " << p.second << endl;
 return 0;
}
```

Sets

- A set is a container that stores unique objects. If a set already contains a certain element, adding that element again does nothing. Sets are ordered;
- adding/removing elements takes logarithmic time which is relatively slow;
- searching also takes logarithmic time, this is as fast as a research can get.

Sets

[C++] Sets

```cpp
#include <iostream> //for cout and cin
#include <set> //for sets
using namespace std;

int main(){
 set<int> s; //set with base type int
 s.insert(7);
 s.insert(1);
 s.insert(5);
 for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
 {//traverse with iterator
 cout << *it << endl; //prints 1,5,7
 }
 if(s.count(8)) cout << "The set contains the number 8" << endl;//search in the set
 return 0;
}
```

Pointers

- It is possible to visualize the memory as a discretized line where each point represents one bit:
- groups of different number of bits represent different types of data:
  - 8 bits: boolean type;
  - 32 or 64 bits: integer type.
- the position in the memory is automatically assigned for each variable that is defined.

Pointers

- C++ permits to know the position (the address) of the first bit allocated for each variable;
- the address of the variable can be saved in a new type named pointer that is able to hold the address of the variable it points to;

[C++]

```cpp
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
yPtr = &y; // this statement assigns address of y to yPtr
```

- the address operator (&) is a unitary operator that obtains the memory address of its operand.

Pointers

- The * when appears in declaration is not an operator but it indicates that the variable being declared is a pointer:

[C++]

```
int * p1;
```

- The * operator commonly referred to as DEREFERENCING operator, returns a ”synonym” for the object to which it points to:

[C++]

```
int *p1; //pointer declaration
int a = 7;//variable declaration
p1 = &a;
cout << *p1 <<endl;//the output is 7
cout << p1 <<endl;//the output is a memory address (e.g.0013F580)
```

Basic pointer manipulations

[C++] Basic pointer manipulations

```cpp
#include <iostream>
using namespace std;
int main()
{
int *p1, *p2;
p1 = new int;
*p1 = 42;
p2 = p1;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
*p2 = 20;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
p1 = new int;
*p1 = 100;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
return 0;
}
```

Basic pointer manipulations

[C++] Basic pointer manipulations: solutions

```
*p1 = 42
*p2 = 42
*p1 = 20
*p2 = 20
*p1 = 100
*p2 = 20
```

Exercise

[C++] Basic pointer manipulations

```cpp
#include <iostream>
using namespace std;
int main()
{
int a;
int *aPtr;
a = 7;//assigned 7 to a
aPtr = &a;//assigned the address of a to aPtr
cout << "the address of a is: " << &a << endl; //0012F580
cout << "the value of aPtr is: " << aPtr << endl; //0012F580
cout << "the value of a is: " << a << endl; //7
cout << "the value of *aPtr is: " << *aPtr << endl; // 7
//The content of the memory address the pointer points to
return 0;
}
```

Exercise

Write a program that prints out:

- the address of a double variable a (you have to define it);
- the value of *aPtr;
- the address of a;
- the content of aPtr;

Exercise

[C++] Basic pointer manipulations

```
#include <iostream>
using namespace std;
int main()
{
double a;
double *aPtr;
a = 8.5;//assigned 8.5 to a
aPtr = &a;//assigned the address of a to aPtr
cout << "the address of a is: " << &a << endl; //memory address
cout << "the content of aPtr is: " << aPtr << endl; //memory address
cout << "the value of a is: " << a << endl; //8.5
cout << "the value of *aPtr is: " << *aPtr << endl; // 8.5
//The content of the memory address the pointer points to
return 0;
}
```

Pass by value, reference or pointer

- There are three ways in C++ to pass arguments to a function: by value, by reference with reference arguments, by reference with pointer arguments.
- When an argument is passed by value a copy of the argument's value is made and passed to the called function.
- With pass by reference with reference arguments and pass by reference with pointer arguments methods we can eliminate the pass-by-value overhead of copying large amount of data since caller gives the called function the ability to access the caller's data directly.

To recap

Write a program that prints out the cube of an integer number (e.g. int number = 5;). For the cube calculation you have to implement a function named cubeByValue that takes as an input the integer number defined before (pass by value) and returns the cube.

Pass by value, reference or pointer

[C++] Solution

```cpp
#include <iostream>
using std:: cout;
using std::endl;

int cubeByValue( int );//function definition

int main()
{
int number = 5;
int result;
cout << "The original value of number is: " << number;
result = cubeByValue( number ) ; //pass number by value to cubeByValue
cout << "The result is: " << result << endl;
return 0;//successful main termination
} //end main
//calculate and return cube of integer argument
int cubeByValue( int n )
{
return n*n*n;
} //end function cubeByValue
```

Pass by value, reference or pointer

- There are three ways in C++ to pass arguments to a function: by value, by reference with reference arguments, by reference with pointer arguments;
- when an argument is passed **by value**, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller;
- with **by reference with reference arguments**, the caller gives the called function the ability to access the caller's data directly and to modify that data if the called function chooses to do so;
- **pointers** can also be used to modify one or more variable in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.

Pass by value, reference or pointer

[C++] Pass by reference with a pointer argument

```
#include <iostream>
using std:: cout;
using std::endl;

void cubeByReferenceP( int *);

int main()
{
int number = 5;
cout << "The original number of number is: " << number;
cubeByReferenceP( &number ); //pass number address to cubeByReferenceP

cout << "The new value of number is " << number << endl;
return 0;
} //end main
//calculate cube of *nPtr; modifies variable number in main
void cubeByReferenceP( int *nPtr )
{
(*nPtr) = (*nPtr) * (*nPtr) * (*nPtr);
}
```

Exercise

Write a program that prints out the square of an double number. For the square calculation you have to implement a function named squareByPointer which requires a pass by pointer input.

Pass by value, reference or pointer

[C++] Pass by reference with a pointer argument

```cpp
#include <iostream>
using std:: cout;
using std::endl;

void squareByPointer( double *);

int main()
{
double number = 5.6;
cout << "The original number of number is: " << number;
squareByPointer( &number ); //pass number address to squareByPointer

cout << "The new value of number is " << number << endl;
return 0;
} //end main
//calculate square of *nPtr; modifies variable number in main
void squareByPointer( double *nPtr )
{
*nPtr = *nPtr * *nPtr;
}
```

Pass by value, reference or pointer

[C++] Pass by reference

```cpp
#include <iostream>
using std::cout;
using std::endl;

void cubeByReference(int&);

int main() {

        int number = 5;
        cout << "The original number of number is: " << number << endl;
        cubeByReference(number);

        cout << "The new value of number is " << number << endl;
        return 0;

} //end main

void cubeByReference(int& nRef) {

        nRef = nRef * nRef * nRef;
}
```

Summary

- arrays, vectors and strings;
- lists, pairs and sets;
- pointers.