

Introduction to Programming and Computing for Scientists

Lecture 5

Programming in C++: control structures, functions, pointers, arrays, data structures

Katja Mankinen

thanks to Vytautas Vislavicius, Oleksandr Viazlo, Anders Floderus

Lund University

4 October 2017

Today's goals

You will learn

- to use `if` and `if...else` selection statements to choose among alternative actions
- to use `for`, `while` repetition statements to execute statements in a program repeatedly
- how to read and write to text files
- to use data structures to represent a set of data items
- to use C++ standard library class template `vector`
- how to access variables in computer memory
- to declare and use pointers

Control structures - if, else

```
if(condition) {  
    statement;  
}  
else if(condition) {  
    statement;  
}  
else {  
    statement;  
}
```

- if evaluates the condition. If it is true, the statement is executed.
- If it is false, the statement in the optional else clause is executed.
- if and else can be nested.

Control structures - if, else

```
if(condition) {  
    statement;  
}  
else if(condition) {  
    statement;  
}  
else {  
    statement;  
}
```

- if evaluates the condition. If it is true, the statement is executed.
- If it is false, the statement in the optional else clause is executed.
- if and else can be nested.

```
if(5 == 10) {  
    std::cout << "This computer is insane" << std::endl;  
}  
else if(5 == 5) {  
    std::cout << "Everything is fine" << std::endl;  
}  
else {  
    std::cout << "This will never happen" << std::endl;  
}
```

Control structures - for, while

```
for(initialization; condition; statement) {  
    statement;  
}  
  
while(condition) {  
    statement;  
}
```

- The `for` and `while` loops execute statements while some condition is met. They are functionally equivalent.
- Use a `for` loop when you know how many iterations you want to do.
- Use a `while` loop when the number of iterations is unknown, for example if the stopping condition depends on user input.

Control structures - for, while

```
for(initialization; condition; statement) {  
    statement;  
}  
  
while(condition) {  
    statement;  
}
```

- The `for` and `while` loops execute statements while some condition is met. They are functionally equivalent.
- Use a `for` loop when you know how many iterations you want to do.
- Use a `while` loop when the number of iterations is unknown, for example if the stopping condition depends on user input.

```
for(int i = 0; i < 10; i++) {  
    std::cout << "i equals " << i << std::endl;  
}  
  
bool keepGoing = true;  
while(keepGoing) {  
    std::cout << "Still going!" << std::endl;  
    keepGoing = readUserInput(); //This magical function returns true or false  
}
```

Control structures - continue, break

- The `continue` statement is used in loops to skip directly to the next iteration. It works in both `for` and `while` loops.

```
for(int i = 0; i < 10; ++i) {  
    if(i == 5) continue; //5 won't be printed  
    std::cout << "i equals " << i << std::endl;  
}
```

- The `break` statement is used to exit the loop entirely. It works in `for` and `while` loops as well as `switch` clauses (next slide).

```
while(true) {  
    std::cout << "Still going!" << std::endl;  
    if(readUserInput() != true) break;  
}
```

Control structures - switch, do-while

- The switch clause can be used to replace many if statements.

```
switch(variable) {  
  case 0:  
    std::cout << "variable is 0" << std::endl;  
    break;  
  
  case 1:  
    std::cout << "variable is 1" << std::endl;  
    break;  
  
  default:  
    std::cout << "variable is neither 0 nor 1" << std::endl;  
}
```

- The do-while loop works like a while loop, except the condition is checked at the end of the loop instead of the beginning.
- This guarantees that the statement will be executed at least once.

```
bool keepGoing = true;  
do {  
  std::cout << "Still going!" << std::endl;  
  keepGoing = readUserInput(); //This magical function returns true or false  
} while(keepGoing);
```


Namespaces

- A namespace is a place where variables, classes and functions live.
- They can share names as long as they live in different namespaces.
- Typing `std::` in front of all standard functions soon gets tiresome. The `using` keyword allows them to be used without a qualifier.
- If you use an entire namespace, beware of collisions (e.g `std::count` exists).

```
#include <iostream> //For cout

using std::cout; //Now we don't have to type std::cout. Just cout will do.
using namespace std; //Like the above but for everything in the std namespace

namespace first {
    int a = 10;
}

namespace second {
    int a = 20;
}

int main() {
    cout << first::a << endl; //Will print 10
    cout << second::a << endl; //Will print 20
    first::a = 30;
    std::cout << first::a << std::endl; //Will print 30. Using std:: still works.
    // std::cout << a << std::endl; this would give an error: a is not declared
}
```

Example

Temperatures in rainy Lund were measured from Monday to Sunday. Write a C++ program that takes in the temperatures as a user input, and displays and calculates the average temperature of that week.

Example

Temperatures in rainy Lund were measured from Monday to Sunday. Write a C++ program that takes in the temperatures as a user input, and displays and calculates the average temperature of that week.

```
Set total temperature to zero
Set day counter to one

While day counter is less than or equal to seven
  Prompt the user to enter the next temperature
  Input the next temperature
  Add the temperature into the total temperature
  Add one to the day counter

Set the temperature average to the total temperature divided by seven
Print the temperature average
```

Example

```
#include <iostream>

using namespace std;

int main(){

    int total; // sum of temperatures
    int dayCounter; // number of the day to be entered next
    int temperature; // temperature value entered by the user
    int average; // average of temperatures

    total = 0;
    dayCounter = 1;

    while (dayCounter <= 7){ //how would you do this with a for loop?
        cout << "Enter temperature: "; //why not std::cout?
        cin >> temperature;
        total = total + temperature;
        dayCounter = dayCounter + 1;}

    average = total/7; // what might be problematic here? how to fix it?

    cout << "Average temperature is " << average << endl;
    return 0;
}
```

A word of warning

```
#include <iostream>

using namespace std;

int main()
{
    int loopCount;
    cout << "Enter loopCount: ";
    cin >> loopCount;
    while (loopCount > 0){
        cout << "Only " << loopCount << " loops to go!\n";
    }
    return 0;
}
```

- What is the problem in the example above?
- How would you fix it?

A word of warning

```
#include <iostream>

using namespace std;

int main()
{
    int loopCount;
    cout << "Enter loopCount: ";
    cin >> loopCount;
    while (loopCount > 0){
        cout << "Only " << loopCount << " loops to go!\n";
    }
    return 0;
}
```

- What is the problem in the example above?
- How would you fix it?

```
for (int i = 1; i <= loopCount; i++){
    cout << "We've finished " << i << " loops\n";
}
// or:
while (loopCount > 0){
    cout << "Only " << loopCount << " loops to go!\n";
    loopCount = loopCount - 1;
}
```

I/O - Reading and writing files

- Reading and writing files is done using the `ifstream` and `ofstream` classes defined in the `fstream` library. The following program reads numbers from a file (`input.txt`) and prints the sum to another file (`output.txt`).

```
#include <iostream> //For cout
#include <fstream> //For ifstream and ofstream

int main() {
    std::ifstream inFile("input.txt"); //Name of the file to read from
    if(!inFile) {
        std::cout << "Error: could not read from file input.txt" << std::endl;
        return 1; //A nonzero return value indicates failure
    }
    double variable = 0.;
    double sum = 0.;
    while(inFile >> variable) { //Read numbers until we hit the end of file
        sum += variable;
    }
    inFile.close();

    std::ofstream outFile("output.txt");
    if(!outFile) {
        std::cout << "Error: could not write to file output.txt" << std::endl;
        return 1; //A nonzero return value indicates failure
    }
    outFile << sum << std::endl;
    outFile.close();
    return 0;
}
```

Containers. Arrays

- How to deal with a collection of data, such as a list of measurement values or a list of names?
- An array is a fixed-size sequential container.
- To refer to a particular location or element in the array, you can specify the name of the array and the position number of the particular element:
`int t[8] = {20, -17, 4, 16, 12, 16, 8, 5};`
- Multidimensional arrays: `type name[size1][size2]...[sizeN];`
→ `int chessBoard[8][8]`

	t
t[0]	20
t[1]	-17
t[2]	4
t[3]	16
t[4]	12
t[5]	16
t[6]	8
t[7]	5

Containers. Arrays

- Try to avoid using arrays in C++. Use vectors instead (next slide). Comments to the code below contain possible pitfalls of using arrays.
- Arrays allocated on the heap are deleted with the `delete[]` operator.

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

int main() {
    const int length = 10; //The length must be known at compile time
    int arr[length]; //This array is fixed-size
    int input;
    int pos = 0; //An array doesn't know its own size or how many elements it contains
    while(cin >> input) {
        arr[pos] = input;
        if(pos == length) break; //Remember that the array can't grow, so this is our limit
        ++pos; //We have to keep track of the position
    }
    for(int i = 0; i < pos; ++i) cout << arr[i] << endl; //Easy to go out of range
    return 0;
}
```

Vectors

- A vector is a sequential container that can change size dynamically.
- It is a *template class*. The vector type must be defined at compile time.
- Vectors are fast at element access and insertion/removal at the end.

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

int main() {
    vector<int> vec; //Create a vector with base type int
    int input;
    while(cin >> input) vec.push_back(input); //Store each input
    for(size_t i = 0; i < vec.size(); ++i) cout << vec.at(i) << endl; //Print them back
    return 0;
}
```

Vectors

- A vector is a sequential container that can change size dynamically.
- It is a *template class*. The vector type must be defined at compile time.
- Vectors are fast at element access and insertion/removal at the end.

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

int main() {
    vector<int> vec; //Create a vector with base type int
    int input;
    while(cin >> input) vec.push_back(input); //Store each input
    for(size_t i = 0; i < vec.size(); ++i) cout << vec.at(i) << endl; //Print them back
    return 0;
}
```

- Use `at` to access individual elements. It's also possible to use `[]`. **Try to avoid this!** There is no bounds checking at run time. Your bugs will go unnoticed.

```
vector<int> vec; //Create an empty vector
cout << vec[3] << endl; //Index is out of bounds. Your program will happily print garbage
cout << vec.at(3) << endl; //Using at produces an error at run time, exposing your bug
```

Strings

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

```
#include <iostream> //For cout and cin
#include <string>
using namespace std;

int main() {
    string str("It's 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; // substring; (string.substr(int pos = 0, int n =
        string::npos) const;)
    return 0; //It's dangerous to take this!
}
```

Reading out line by line using string

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

```
#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;
}
```

Pointers - motivation

- How variables are stored in computer's memory?

Pointers - motivation

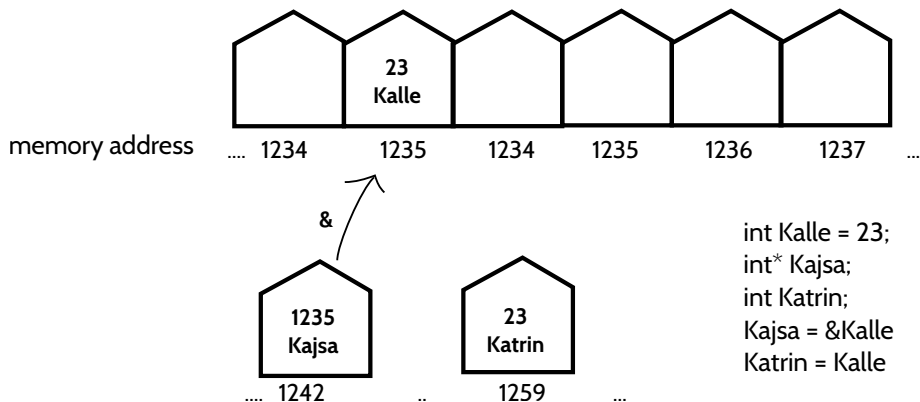
- How variables are stored in computer's memory?
- Variables are stored in memory cells inside the computer's memory. Cells have unique addresses.
- When you refer to a variable by name in your code, the computer looks up the address that the variable name corresponds to, goes to that location in memory, and retrieves or sets the value it contains.

Pointers - motivation

- How variables are stored in computer's memory?
- Variables are stored in memory cells inside the computer's memory. Cells have unique addresses.
- When you refer to a variable by name in your code, the computer looks up the address that the variable name corresponds to, goes to that location in memory, and retrieves or sets the value it contains.
- Manipulating the memory addresses of data can be faster and more efficient than manipulating the data itself.
- If you have a large data structure, and you pass it by value, the computer has to push a copy of it onto the stack. This wastes both time and stack-space!
- If you pass over only the address, you don't need to make a big copy and you only push an address onto the stack.
- Very common use case: a pointer to a resource (file, database, histogram...) that was made somewhere else by some other code, and you have to interact with it for a while
- Or a pointer to a resource that is shared between multiple objects: each object holds a pointer to the resource rather than always copying it.

Kalle's home street - very simplified example

- Example on whiteboard



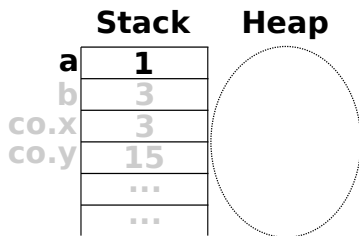
The stack and the heap

- The memory available for a program to use (at least as far as we're concerned) is made up of two areas - The stack and the heap.
- The stack is a small (megabytes), fixed size chunk of memory for local variables. All examples so far have used only the stack.
- When a variable on the stack falls out of scope, it is deallocated. You don't have to worry about memory management with the stack.
- The stack is small, so it overflows if you put too many things on it. But don't worry - This typically only happens due to bugs (e.g an infinite loop).

```
#include "coords.h"

void makeCoordinates(int b) {
    coords co(b, b*5);
}

int main() {
    int a = 1;
    makeCoordinates(a + 2);
    //Grayed out variables have now been deallocated
    return 0;
}
```



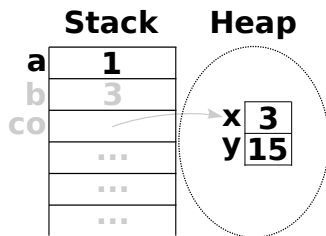
The stack and the heap

- The heap is a large pool of memory that can grow dynamically.
- To put a variable on the heap, create it with the `new` operator. This operator returns a *pointer* through which the variable is accessed.
- A pointer is really just an integer. The number corresponds to a memory address. The pointer *points* to that memory.
- Variables on the heap are never deallocated automatically. The memory must be freed manually using the `delete` operator.
- The pointer itself is on the stack and is deallocated automatically.

```
#include "coords.h"

void makeCoordinates(int b) {
    coords* co = new coords(b, b*5);
}

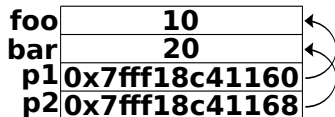
int main() {
    int a = 1;
    makeCoordinates(a + 2);
    //Grayed out variables have now been deallocated
    return 0;
}
```



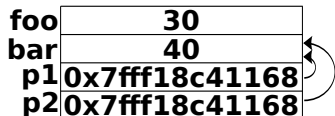
Pointers and references

- A pointer can point anywhere in memory, both the stack and the heap.
- To declare that a variable is a pointer, put an asterisk (*) after its type.
- To get the memory address of a variable, use the reference operator (&).
- If you have a pointer and you want the value that the pointer points to, use the dereference operator (*). That's right - The asterisk has *two* uses!

```
int foo = 10; //Two regular variables
int bar = 20;
int* p1; //Two pointers to int
int* p2;
p1 = &foo; //p1 points to foo
p2 = &bar; //p2 points to bar
```



```
*p2 = 30; //bar = 30
*p1 = *p2; //foo = bar
p1 = p2; //p1 now points to bar
*p1 = 40; //bar = 40
```



- To access members of a class via pointer, use the arrow (->) operator.

```
betterCoords a(1, 1); //Regular object
a.SetCartesian(2, 2); //Access with dot
betterCoords* b = new betterCoords(1, 1); //Pointer to object
b->SetCartesian(2, 2); //Access with arrow. This is the same as (*b).SetCartesian(2, 2)
```

Pass by value, reference or pointer

- When calling a function, you are really passing *copies* of all the arguments.
- If you want to change the passed values, you must use references or pointers.

```
int x = 1;
int y = 2;

void swapByValue(x, y); //This will NOT swap the values!
void swapByReference(x, y); //This will work. Using references is recommended.
void swapByPointer(&x, &y); //This will work, but don't use pointers unless necessary.
```

```
void swapByValue(int a, int b) { //a and b are copies of x and y
    int temp = a; //Whatever we do here has no effect on the original x and y
    a = b;
    b = temp;
}
```

```
void swapByReference(int& a, int& b) { //a and b are references to x and y
    int temp = a; //For all intents and purposes, they ARE x and y
    a = b;
    b = temp;
}
```

```
void swapByPointer(int* a, int* b) { //a and b are pointers to x and y
    int temp = *a; //Not safe - What if they are NULL pointers? Use references instead.
    *a = *b;
    *b = temp;
}
```

Basic pointer manipulations

```
#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

```
#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;
}
```

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

Summary

- Control structures: `if-else`, `for`, `while`, `continue`, `break`, `switch`, `do-while`
- Containers: arrays and vectors
- Stack and heap
- Pointers

Homework 1

Homework 1 instructions are at Live@Lund. You'll be a rocket scientist!

The deadline for submitting your solution is **22 October**!

Remember: homework is **mandatory**! But also remember: if you get stuck, you can submit incomplete homework and always ask for help from me.

Read the following chapters from the course book "Problem solving with C++" by W. Savitch:

- 2.4 Simple flow of control
- 3.2 Multiway branches
- 3.3 More about C++ loop statements
- 6 I/O Streams as an introduction to objects and classes
- 7 Arrays
- 8 String and vectors
- 9 Pointers and dynamic arrays

Next Lecture

- Classes
- Inheritance
- Polymorphism
- The `const` keyword
- Type Casting
- Operator overloading
- Templates

Lists, Pairs

- 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.

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

int main() {
    list<int> lst; //List with base type int
    lst.push_back(10); //Insert some elements, then 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;
}
```

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

```
#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 search can get!

```
#include <iostream> //For cout and cin
#include <set>
using namespace std;

int main() {
    set<int> s; //Set with base type int
    s.insert(7); //Add some elements. The order in which they are added doesn't matter.
    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;
}
```

Maps

- A map is an associative container that stores key/value pairs. A key can not be inserted twice, but the value of an existing key can be changed.

```
#include <iostream> //For cout and cin
#include <string>
#include <map>
#include <utility> //For make_pair
using namespace std;

int main() {
    map<string, int> pBook; //Map associating strings to ints. It's a phone book!
    pBook.insert(make_pair("Reginald", 123)); //Pairs can be inserted in various ways
    pBook.insert(pair<string, int>("Marmaduke", 456));
    pBook["Bobby Floyd"] = 789;

    map<string, int>::iterator it = pBook.find("Bruce Lee"); //How to search a map
    if(it != pBook.end()) cout << it->first << "has number " << it->second << endl;
    pBook["Reginald"] = pBook["Jim Bob"]; //Beware of using [] - Jim Bob is now in the book

    for(map<string, int>::iterator it2 = pBook.begin(); it2 != pBook.end(); ++it2) {
        cout << it2->first << " - " << it2->second << endl; //Print everyone in the book
    }
    return 0;
}
```

```
Bobby Floyd - 789
Jim Bob - 0
Marmaduke - 456
Reginald - 0
```