[PRAKTISCHE ASPEKTE DER INFORMATIK – WS 13/14]

All elemental steps that will get you started for your new life as a computer science programmer.
Week B – Pointers, References, Inheritance

Pointers and References
Why use pointers at all?
Inheritance in C++
Assertions
Assignment: Build your own 3D house

Week B: Pointers, References and Inheritance
This week we’ll cover more advanced concepts of C++: pointers and references. You might not be familiar with these, as languages like JAVA hide these concepts from you.

We’ll also talk some more about Object Oriented programming and a useful technique called assertion.
Pointers and References

Easy to spot: & and *

References are safer to use
  - always use the `const` keyword if possible

Pointers give you more freedom, but you have to allocate and deallocate memory yourself
  - New and delete operators

Pointers and References

So far, we did not use any Pointers or references. When you code small programs that only handle small amounts of data, you do not have to worry too much about using them, anyhow.

In C++, whenever you pass an input parameter to a function, C++ implicitly copies this parameter for you, so that local changes on this variable do not affect the piece of code that called this function. With references and pointers you can actually avoid this implicit copy operation, so that functions can alter data that has been passed to them.

We’ll soon see how this is done and why we should do it in a minute.
These functions show how to use references and pointers. float change_value_cp(float float_copy) shows a typical function like you already know it. The parameter float copy is actually copied implicitly by C++, so you can make local changes to it without actually changing its original content. Note that although you may increase its value locally, the original value stays the same.

float change_value_ref(float& float_reference) accepts a reference as its input. You call it like you called change_value_cp(). The difference is that when passing a reference (instead of an implicit copy), local changes of float_reference also affect the original variable that has been used as the input parameter. You can avoid these changes by using const references, as in float change_value_const_ref(const float& const_float_reference). The code would not compile if you try to change the value of const_float_reference. You may wonder why you should use a const reference at all, but it may be very desirable to avoid an implicit copy at all, i.e. if the data passed as the input argument is very large (maybe it’s a bitmap image or the contents of a video file). In this case, copying data all the time would require a lot of system resources.

The next two examples demonstrate the usage of pointers. Pointers are not data containers themselves; they merely store the actual location in your systems memory where your data is stored. E.g., a float* (i.e. a pointer to a float) only stores the location where a float is stored. You may access the content of a pointer (e.g., the float variable) via the (*...) expression, e.g. with (*float_pointer) in change_value_ptr().

```c++
float change_value_cp(float float_copy)
{
    return ++float_copy;
}

float change_value_ref(float& float_reference)
{
    return ++float_reference;
}

float change_value_const_ref(const float& const_float_reference)
{
    return const_float_reference+1;
}

float change_value_ptr(float* float_pointer)
{
    return ++(*float_pointer);
}

float change_value_const_ptr(const float* const_float_pointer)
{
    return (*const_float_pointer)+1;
}
```
A new concept that comes with pointers is that you have to explicitly allocate and deallocate memory for them. With the new operator, C++ is told to reserve an appropriate amount of memory for your variable and return the address to this chunk of memory for you.

In the end, you always have to delete the space used by your variable. If you delete the space and try to access its content afterwards, you may (or may not) run into weird problems, as C++ is free to use this space for other purposes. If you access the memory outside of the allocation space, the program might crash too! This is called a out-of-bound access.

```cpp
float* my_ptr = new float(0.0);
// do stuff with your new pointer
delete my_ptr;

// allocates 4 floats consequently in memory
float* my_ptr = new float[4];
// values are uninitialized, so initialize them ...
float f = float[1]; // access the second float in the
float e = float[5]; // !this will crash the program!
delete[] my_ptr;
```
A useful coding technique when working with pointers is to immediately write the invocation of delete, after declaring or allocating the pointer. This might save a lot of time later, that otherwise would be spend on looking for memory leaks.

If the pointer should be defined, but no memory should be allocated for it (e.g. because it should only be used to point to another variable) initialize the pointer to NULL, to signalize that it has no valid memory to point to.
Accessing memory locations

When a function expects a “normal” variable or a reference as a parameter, you can just pass your variable as the input parameter. When a pointer is expected, but you do not have one, you may access the memory location of your variable with the `(…)` operator. But be careful: this memory location will be deallocated automatically by C++ when leaving the scope where it was created. The current scope always ends after the next `}` bracket is reached in the code. You may access the actual content of a pointer with the `(…)` expression.
**Why use pointers at all?**

YOU can decide when your variables live and die.
You can control memory allocation
Pointers can be shared – but beware!
You can pass NULL-pointers (like in JAVA)
And there are “smart pointers”

**Why use pointers?**

It may seem superfluous to use pointers at all, you can do a lot with just using “regular” variables and references.

Still, it is very handy to use pointers in some scenarios. Pointers can be very useful, for modularizing your code. When a single object should be shared by different other object, they can simply share the pointer to the same memory location, you just have to make sure that none of these objects deletes the object without notifying the other ones. Additionally, when you want to control how long your variables should “live”, pointers are your weapon of choice.
HEAP and STACK?

By default, Memory is allocated on the stack.

- Memory is deallocated when you leave current scope
- You can access the memory address, but be careful with it!

If you use new, objects are stored on the heap

- You have to clean up with delete

For a thorough explanation, you may check out

Inheritance in C++

Inheritance in C++:

- Classes may inherit from multiple other classes (polymorphism)
- Classes can be abstract (like Interfaces in JAVA!)
- Eventually, you need to define your own constructors, assignment operators and destructors explicitly
- Virtual classes allow method overriding

Inheritance in C++

If you have used inheritance in JAVA, you will see that inheritance in C++ works very similar. The main difference is that you can inherit from multiple classes. This can be quite handy sometimes, but may also cause a lot of problems when things get too complicated. A good way of using inheritance wisely is to declare purely abstract classes, i.e. classes that only declare functions but do not implement them.
This is an example of a virtual, abstract class: functions are only declared, but not defined. Here, no constructor exists, but it could be possible to use one. It is also allowed to implement some of the methods, but as long as the class contains one abstract method, it cannot be instantiated.

If you put a

= 0;

expression after the declaration of a function, you tell C++ that this method is not implemented in the current class. The keyword virtual tells the C++ Compiler that the method that is executed is determined during run-time of the program and depends on the actual class type of the instance. If you use non-virtual functions, C++ decides which method to call during compile-time.

A nice overview of virtual functions in different programming languages can be found at: http://en.wikipedia.org/wiki/Virtual_function
The class `Tree.h` does actually implement the abstract functions. Note that we need to explicitly implement a destructor, copy constructor and assignment operator because we dynamically allocate memory in the instance.

Note the virtual destructor. When a destructor of a class is invoked, it always invokes all the “down” in his line of inheritance. The keyword virtual here, makes sure that the destructor of the actual class type is invoked. Consider this example where class B is derived from A:

```cpp
B* b = new B();
A* a = b; // This works because B is also from type A

// This will only invoke the destructor of class A if the // destructor is not declared as virtual!
delete a;
```
Creating and accessing instances

Instances of classes can be created like this

```cpp
class A
{
    public:
        A();
        A(const int);
        void foo();
    }

    A a0(5);
    A a1 = A();
    A* a_ptr = new A(42);

    To access pointer instances use this
    (*a_ptr).foo();
    a_ptr->foo();
```

If you access members or functions of a class, you may alternatively use the -> symbol. These two expressions are equivalent

```cpp
(*myObject).doStuff();
myObject->doStuff();
```
The Class Conifer also "overwrites" a few functions, but only three of them. The other functions are inherited from the base class Tree.

Notice here, that the destructor is not defined virtual, but it already is, because the destructor is virtual in the base class. If a class is derived from another class, all methods which are inherited and are virtual in the base class, are also virtual in the derived class, even if it is not explicitly stated.
The Rule of Three

If a class defines one of the following it should probably explicitly define all three:

- Destructor
- Copy constructor
- Assignment operator

Understand shallow copy vs. deep copy


The Rule of Three is a useful C++ programming guide line. Whenever your class needs to define a destructor, which is always the case if you dynamically allocate memory in the instance somewhere, you always need to implement a explicit copy constructor and assignment operator.

Assertions

Make your application crash ...
...but are very useful!

```c
#include <assert.h>

void foo(float* ptr)
{
    assert(ptr != NULL);
    // Do something with ptr ...
}
```

It is better when an application crashes in front of you than in front of a user!

Assertions
Whenever you code an algorithm based on certain assumptions, and you want to make sure they are fulfilled, you can declare an assumption. If you compare two vectors and assume that they are of the same length, use an assertion:

```c
assert(vector1.size() == vector2.size());
```

If you compute a probability value, you can assume that its value lies between 0 and 1:

```c
assert (probability >= 0.0 && probability <= 1.0);
```

By using assertions, you make sure that mistakes in your code get punished immediately, instead of making your program crash a few hundred lines of code later.

Assertions don’t affect the performance of your application if done right. Usually, you disable the assertions in your final release build (done via compiler flags) and so the assertion code is not included in the final binary.
Assignment_stub: Tree.cpp

#include <assert.h>

// Add a child object to this object
void Tree::addObject(BuildingObject* obj)
{
    std::cout << "(Tree) no objects can be inserted to a tree!" << std::endl;
    assert(false);
}

It may also be a good idea to give a more detailed output before you let an assertion stop your program.