Advanced Programming

course notes by Kees van Kempen (ru@keesvankempen.nl)

Notes are going to be quite brief, with a sparkle of personal comments. As there only was one whole lecture, the rest is gonna be something of a summary of useful facts.

The course seems to follow Wouter Verkerke's course on *Introduction to C++ and Object Oriented Programming* from 2006 with updates in 2015 called v55, although I found v60 from 2018 somewhere on a NIKHEF webpage.

I found out the hard way that copying code does not work. The rendered LaTeX pdf files sometimes changes hyphens and quotation marks to different characters depending on their place. BAH.

See notes.pdf for a pdf of these wonderful notes.

I figured out http://cpp.sh/ exists. It is really useful for quick debugging of snippets.

Lecture 1 (2021-11-08)

Administrative

- Use Gitlab to hand everything in.
- Seem to be using g++ with c++11 standard.
- Look into debugging tool Valgrind.
- Maybe try to explore using VScode instead of vim.
- There is no fixed exam date (which Frank finds a pity). Handing in is flexible, "at least until the end of the year". He will, however, try to fix an exam date (as there is one person who likes the idea).
- Failure is not an option. Retries until the student gets it right is the idea.

Substance

He asks who knows what a pointer is. "Quite a few do not." Pointers are references to memory addresses. (A reference is something else, though.) The number of memory addresses is limited by the CPU arch. Here comes the difference between 32 and 64 bits. The length of the memory address is 32 bits vs 64 bits. 32 bit computers can often only handle some 4 GB of RAM. He asks us to familiarize ourselves with the concept, but is not giving a thorough lecture about them.

Wrapping up

He wraps up after the 45 minutes with Walter (TA) giving some explanation about that he is writing some introductory thing and that Discord is our communication channel of choice. He and Frank encourage us to ask questions in there so other can also benefit from it. They also recommend looking at the C&CZ wiki for information about, e.g., Gitlab and ssh.

1 The basics of C++

- Some basics on compilation using `g++`.
- Basic syntax, declaration of objects, keyword `const`, difference between declaration and initialization, auto declaration.
- Arrays
  - Can be setup as `Type name [size]` or, for multiple dimensions, `Type name [size1][size2]...[sizeN]`.
  - Is just a sequence of memory allocations of the same size.
- References: aliases for the same type and memory location.
  - `Type& name = othername`
  - `&name` returns the pointer to `name`.
- Pointer: object that holds a memory address with associated type.
  - `Type* ptr = othername`
  - `*ptr` returns a reference to the thing `ptr` points at, i.e. `othername`.
- Note that the `*` and `&` keys have "opposite" effect depending on placement.
- For an array `Type a[size]`, `a` is a pointer to `a[0]`.
- For the same array, `a[i] == *(a+i)`.
- In general, type is known at compile-time, but unknown at run-time.
- Something is said about scopes and memory.
  - See http://www.gotw.ca/gotw/009.htm, note it is pre-standard.
  - Most variables are defined at compile-time, and memory is statically allocated.
  - Allocation can be in the
    - *constant data area*, for constants,
    - *stack*, for automatic variables, such as defined at compile-time, used in functions, cleaned up after returning of the function,
    - *free store/heap*, for dynamically allocated memory, note memory leakage, allocation through `malloc()` and keyword `new`, e.g..
- Dynamic memory allocation is possible in the `free store`.
  - `Type* ptr = new Type` allocates memory, returns pointer.
  - Memory is only freed if it is done explicitly by the the program. (Something about memory ownership, cleaning up bla.)
  - `delete ptr` frees the memory.
  - Terminating the program lets the kernel clean everything up.

2 Files and Functions

- Something very basic on syntax of functions.
- Declaration and definition of functions are, like with variables, separate things. You can declare one early (forward declaration) on and define it later. (Do you feel header files? I do.)
  - Declaration does not need to name the arguments.

- Three ways to pass arguments to function `Type f(args)`:
  - pass by value: `f(Type a)` gives a copy to the function;
  - pass by reference: `f(Type& a)` gives a reference, chance affects the original;
  - pass by `const` reference: `f(const Type& a)` gives a reference, disallows changing `a`, creates no copy, so is memory efficient;
  - pass by pointer: `f(Type* a)` gives a pointer to `a`.
- `int main(int argc, const char* argc[])` provides command line args.
- Default arguments are a thing: `f(Type = value)`. Argument is then optional. Must be provided in initial declaration!
- Overloading is a thing: same function name, different signature.
  - Overloading by return type is *not* a thing.
- I figured out during ex2.1 that an alias `using cstring = const char*;` is a very suitable substitution, and feels quite natural.
- Arrays are weird. Just treat them like pointers most of the time.
  - Look at a multidimensional array to agree with me that they are weird. Passing an array `int a[n][m]` to a function can be done as:
    - simply a pointer `f(int *p)`
    - as an array `f(int p[][m])`
    - as an n dim array of pointers to m dim arrays `f(int (*p)[m])` (or `f(int *p[m])`)
    - as a pointer to a pointer `f(int **p)`
- Modules are a great way of making code reusable.
  - Declaration and definition should be split, so that declaration can be used in other code to reference the definitions.
  - Header files are a thing. Put the declarations in there, make sure that every compilation call only uses the declaration once, and put the definitions in regular files.
  - Under the hood, the header files `.hh` are of the same type as `.cc` code, but it is all convention.
  - To make sure the declarations are only interpreted once, we use include guards.
    - A more modern way is using `#pragma once`.

```
// banaan.hh
#ifndef BANAAN_HH
#define BANAAN_HH
void pellen(int);
#endif

// banaan.cc
#include "banaan.hh"

void pellen (int snelheid) {
  // ...
}
```

```
// main.cc
include "banaan.hh"

int main() {
  pellen();
}
```

- Namespaces can be used to scope code. They are invoked by `namespace name { /* functions, classes etc */ }`
- Things from namespaces can be used in multiple ways:
  - by `using name::function;` (`::` = scope resolution operator)
  - by `using namespace name;`
- Back to modules, or better yet: static libraries.
  - Modules do not need a `int main()` as that is not executed.
  - Modules are compiled by `g++ -c module.cc` so that `module.o` is created, an object file.
  - Objects can be packed using the archive `ar` binary: `ar q libVeryNiceName.a module1.o module2 [...]`
  - Using them then requires the inclusion of the required header files in the code, and compiling using the library: `g++ program.cc -L. -lVeryNiceName`. The `-l` arg provides the library name that should be used in compilation. The `-L.` arg adds directory `.` to the path in which to search for libraries. So yeah, the `lib` prefix is required, and the name of the archive matters.
- There are debugging tips on the slides.
  - `valgrind` is mentioned; I like it for detecting memory leaks.
  - `gdb` is a debugging tool.

3 Class Basics

- This week is mostly about defining your own data types:
  - Classes (`class`) and structures (`struct`) are used to do this.
  - The great benefits are abstraction and encapsulation. Functionality can be split into chunks.
  - Objects are easy to relate to, easy to visualize. We can define properties of certain objects and operations on them.
  - By defining these data types, and treating them as separate module, we can even reuse code.
  - I suspect the notion of inheritance and even polymorphism will come up somewhere in the course, but not yet.
  - Having only used classes in my programming, I am gonna regard classes as structures on steroids.
- Structures

  ```
  struct StructName {
  ```

```
  // We can just define variables and functions in structures.
  char var1;
  int var2 = 3;

  double func1 (double arg) {
    return arg*2 + var2;
  }

  // Or only declare a function.
  void banana ();
};

// This later definition allows one to use header files.
void StructName::banana() {
  std::cout << "Hello i em banannna"  << std::endl;
}
```

- Classes

```
class StructName {
  // Classes allow us to limit access. Normally, the public properties
  // (interface) are defined before the private (implementation) ones,
  // but I copied the above example without much change.
  private:
    char var1;
    int var2 = 3;

  public:
    // We can define a constructor and destructor.
    StructName() { std::cout << "An instance of StructName is created!" << std::endl; }
    ~StructName() { std::cout << "An instance of StructName is deleted!" << std::endl;
    double func1 (double arg) {
      return arg*2 + var2;
    }

    void banana ();
};

void StructName::banana() {
  std::cout << "Hello i em banannna"  << std::endl;
}
```

- FIFO (first in first out) stacks are discussed.
- Classes have constructors and destructors. See the above example.
- On initialization of an object of a
  - structure, it is just created.

  - class, the constructor is ran, on deletion the destructor.

- In a class, `this` is a pointor to the instance you're currently in.
- A member function of a class can be set `const` (`void print() const;`). The function then cannot alter the object.
  - This behaviour can be overwritten by declaring some member `mutable` (`mutable int result;`).
- Static members can be used without initialization of the class. Over all instances of the class, the member variables will be shared.

4 Class Analysis & Design

Object Oriented Analysis and Design

A first shot at decomposing your problem into classes.

- To create object oriented (OO) software, usually three steps are taken:
  1. Analysis (how to divide problem into classes, what should the classes do)
  2. Design (what should the interface to the classes look like)
  3. Programming
- Often, objects in the application domain correspond to natural objects.
- Creating hierarchical ranking of objects using `has-a` relations helps.
- Revisiting choices is important.
- Besides `has-a`, an object can also have a `uses-a` relation, which often results in a pointer.
- Go for a robust design, with intuitive behaviour, go for reusability, don't repeat identical code, avoid complexity.
- Write interface first, then implementation.

Designing the class interface

A story about `const`

One of the most ubiquitous qualifier in C++ is `const`. A variable can be declared constant, a pointer can be declared constant, a reference can be declared constant. We can also declare constant what a variable points to. We can also declare that a class cannot be changed by some member function, thus also using constant. Let's see some examples. Major inspiration is stolen from http://duramecho.com/ComputerInformation/WhyHowCppConst.html.

```
// A constant value
const int var1 = 69;

// A variable pointer to a constant value
const int*    var2 = 69;
int const*    var3 = 69;
```

```
(int const)*  var4 = 69;
(int const) * var5 = 69;

// A pointer to a constant address with a variable value
int* const   var6 = 69;
(int*) const var7 = 69;

// A constant address to a constant variable
int const* const   var8  = 69;
(int const)* const var9  = 69;
const (int const)* var10 = 69;
```

And a completely working example:

```
#include <iostream>

int main()
{
    int* temp = new int;
    *temp = 70;

    // A constant pointer to a variable value
    int* const var1 = temp;

    // A constant address to a constant variable
    int const* const var2 = temp;
    const int* const var3 = temp;

    // A constant value
    const int var4 = 69;

    // A variable pointer to a constant value
    const int* var5;
    int const* var6;

    std::cout << *var2;
    *temp = 71;
    std::cout << *var2;
    // The following is illegal
    //*var2 = 90;
    // But we can alter temp, and thus alter the value of var2
    std::cout << *var2;
    std::cout << std::endl << temp << std::endl << var2;
}
```

I mostly like the conclusion of said website: > Basically 'const' applies to whatever is on its immediate left (other > than if there is nothing there in which case it applies to whatever is $>$ its immediate right).