# C++ Introductory Tutorial

## Part I : Basic Language Features

Institute of Computer Graphics and Algorithms

**Vienna University of Technology**

# Outline

- Two part tutorial:
  - Today:
    - C++ Basics
  - Next week:
    - STL
    - Advanced Topics
    - C++ recipes
      - How to do program common tasks properly
    - Your Questions

# Overview

- Stages of the C++ build process
- Basic syntax
- Declaration vs. Definition (Headers)
- Data types
- Pointer & References
- Important C++ operators
- Global Scope
- Const correctness
- Passing variables
- Stack & Heap Memory
- Classes & Polymorphism

# C Plus Plus

- Developed by Bjarne Stroustrup
  - 1979 Bell Labs
  - Originally named *C with Classes*
- Powerful type-safe language
- Used in
  - Games
  - Embedded Systems
  - High-performance application
  - Drivers, Kernels,...

# C Plus Plus

- C++ is a federation of 4 languages
  - C
    - You can still do any low level C stuff (comes in handy when using C APIs like OpenGL)
  - Object oriented C++
    - Classes, Polymorphism, OOP
  - Template C++
    - Generic programming, template metaprogramming
  - Standard Template Library (STL)
    - A set of standard algorithms and data structures for C++

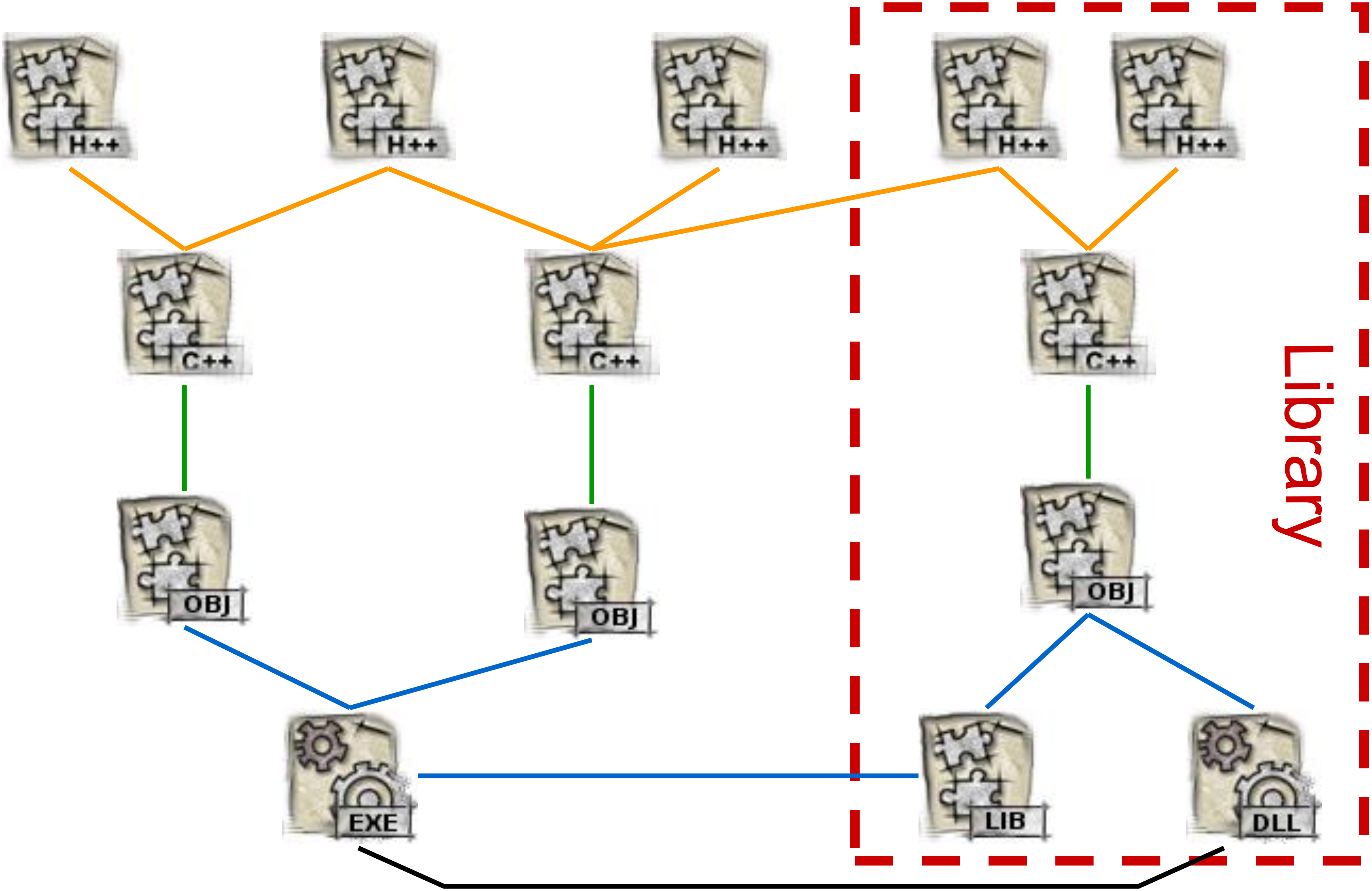
# Header- & Source Files

- Common mechanism for organizing code
- **Header files** store **declarations** and interfaces
  - typical file extensions: *.hpp, *.hh, *.h
- **Source files** store **definitions** and implementations
  - typical file extensions: *.cpp, *.cc
- Implementation details not necessary during compilation as long as interfaces are known

# Stages of the C++ build process

- **Preprocessor** replaces text in files
  - no scope rules whatsoever taken into account
  - Note: human and compiler see different things
- **Compiler** translates source files to object files
- **Linker** merges object files to an executable file

# Header- & Source Files



# Static and Dynamic Libraries

- Object files contain all compiled source code
- Static libraries are essentially object files
  - when linking to a static library, all code that is actually used, is merged into the executable
- Dynamic libraries consist of 2 files
  - *.lib files – contain declarations only and no code; linker knows how much space e.g. a function will need on the stack etc.
  - *.dll files – contain the needed code

# DLLs

- DLLs are not merged into the executable
  - but executable can call code stored in DLL
  - i.e. DLL must reside in the same directory as the executable, or a system directory for DLLs!
- As soon as code from DLL is needed, the DLL is dynamically ("at runtime") loaded into memory
  - system-wide, if the same DLL is used from many processes, its content is loaded into memory only once and can be shared

# LIBs Conclusion

- Include a library‘s header file(s) so that COMPILER knows library variables, function declarations etc.
  - e.g. compiler can perform type checking
- Link to library [*.lib file(s)] so that LINKER can lay out code and calculate jump addresses etc
  - for a dynamic link library don‘t forget to make its code accessible at runtime, i.e. ship *.DLL file(s) with the executable
  - ...it‘s a little different on other platforms than Windows

# Typical Compiler Errors

- Syntax Error
  - misspelled keyword etc.
- Type Error
- Forgot to include a file?
- Wrong include path?

# Typical Linker Errors

- Unresolved reference
  - to variable, function, etc.
- Forgot to link to library files?
- Wrong library path?

# A Simple Example

***cpp_intro.hpp:***

```
#ifndef _CPP_INTRO_HPP_
#define _CPP_INTRO_HPP_

#include <iostream> // for std::cout, std::endl

// usually we'd only declare functions in header files, but
// this way we can see better, what the preprocessor does ☺
void say_hello(void) {
  // “cout” prints to the console
  std::cout << "Hello CG2LU!" << std::endl;
}

#endif //#ifndef _CPP_INTRO_HPP_
```

# A Simple Example

***cpp_intro.cpp:***

```
#include "cpp_intro.hpp"

int main(int argc, char *argv[]) {
  say_hello();
  return EXIT_SUCCESS;
}
```

# A Simple Example

***cpp_intro.cpp after preprocessor-pass:***

```
/*
 *
 * --- MANY LINES OF CODE ----
 * from iostream (a system header file)
 *
 */

void say_hello(void) {
  std::cout << "Hello CG2LU!" << std::endl;
}

int main(int argc, char *argv[]) {
  say_hello();
  return 0;
}
```

# Header Guards

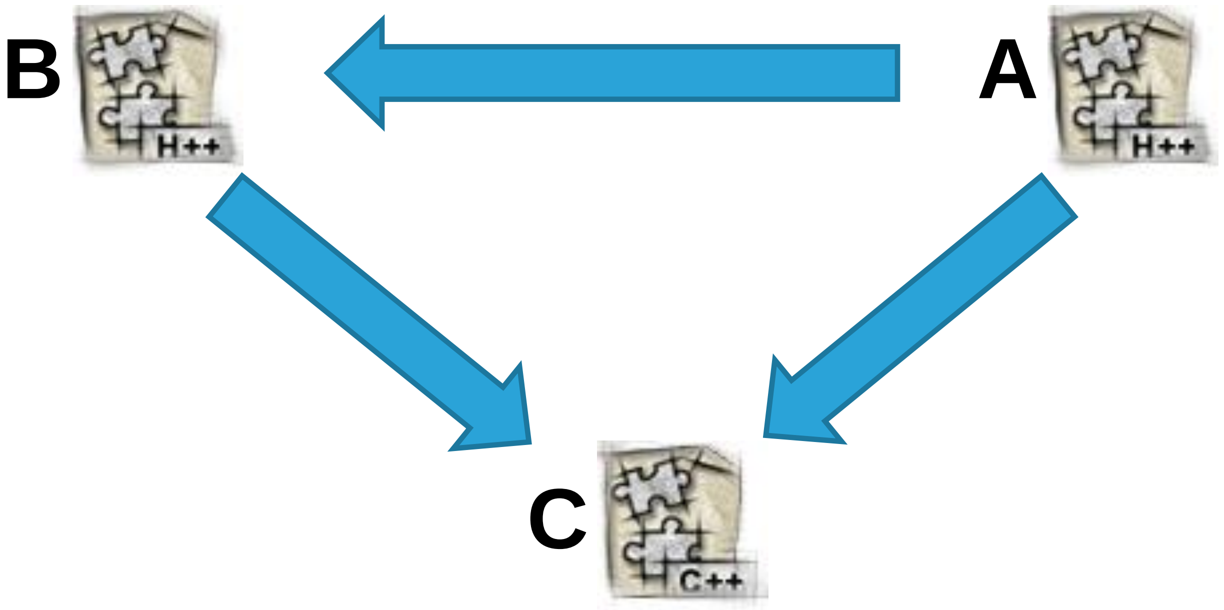
- Large programs tend to include the same header file many times
  - E.g. it is very likely that many source files have a

```
#include <string>
```

- Each time a header file is included in a file, its content is copied into that file
  - i.e. we face the problem of multiple declarations for the same thing
  - compiler doesn‘t like multiple declarations

# Header Guards

- Small multiple inclusion scenario:
  - B.hpp includes A.hpp
  - C.cpp includes A.hpp and B.hpp (which already includes A.hpp itself!)



# Header Guards

- The preprocessor comes to the rescue
  - test, if certain symbol is defined
  - if not, define it and include the file's content
  - if yes, just ignore the whole file

```
#ifndef SOME_TOKEN
#define SOME_TOKEN

     //only included ONCE

#endif //#ifndef SOME_TOKEN
```

# Primitive Data Types

- Basically very similar to Java datatypes
  - **`void`** : „no (specific) datatype“ (e.g. generic pointers, functions returning no value / accepting no parameters)
  - **`char`** : (8 bit) character
  - **`wchar_t`** : wide character (e.g. UNICODE)
  - **`bool`** : boolean
  - **`short, int, long`** : integers
  - **`float, double, long double`** : floating points

# Primitive Data Types

- Some differences to Java:
  - **`unsigned`** datatypes
  - boolean (Java) => **`bool`** (C++)
  - null (Java) => **`NULL`** (C++)
  - no class objects representing primitive types

# Primitive Data Types

- Determining a datatype‘s size in byte
  - e.g. size of an integer

```
sizeof(int)
```

  - check, if code executes in 32 bit (4 bytes) or 64 bit (8 bytes) environment

```
sizeof(void *)
```

# Pointers and References

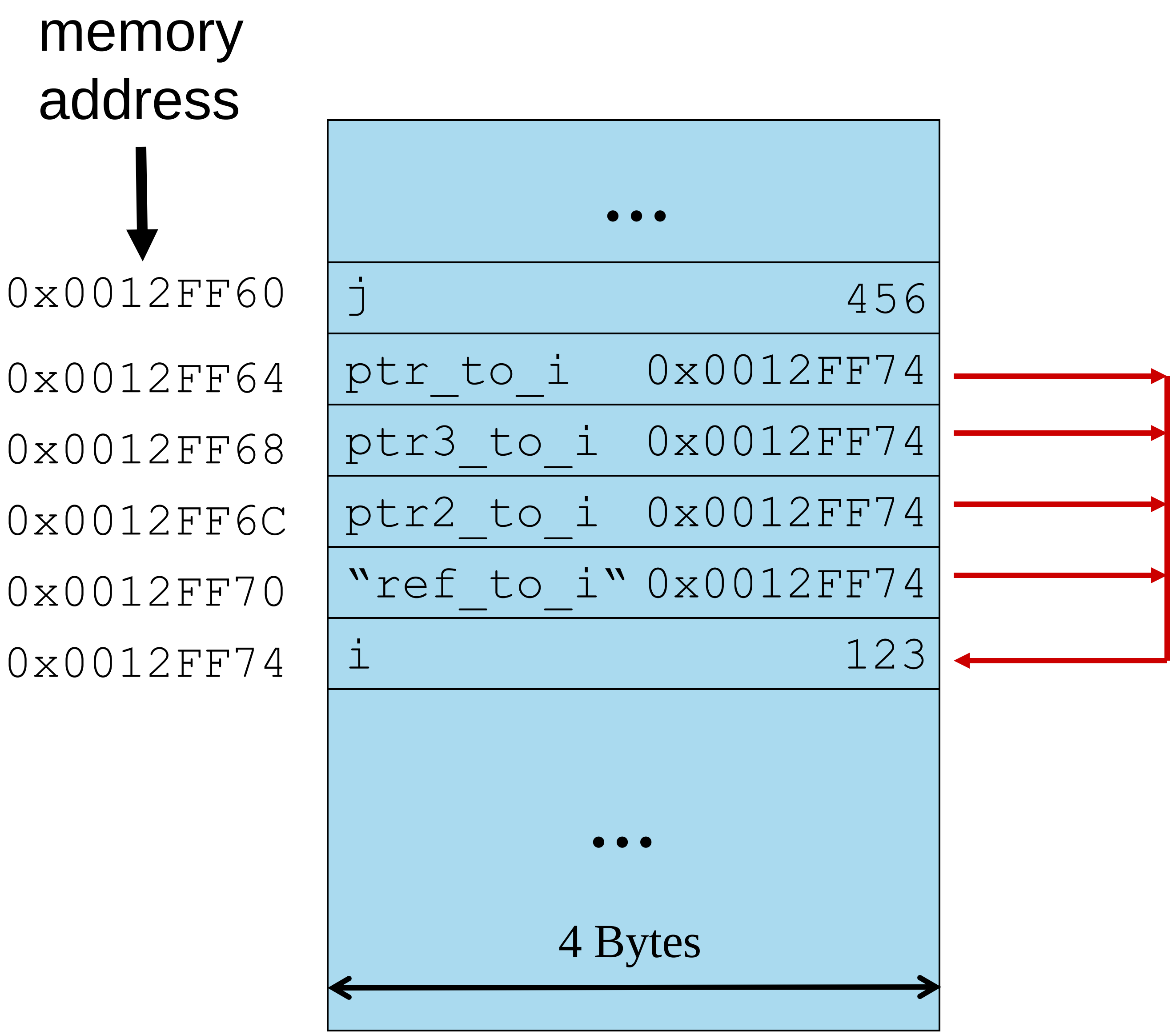
- Pointers
  - Store the address of an object instead of its value/content
  - Need not be initialized
  - We can make a pointer refer to other addresses many times
    - pointer arithmetics
    - the data type determines the step-size in bytes
    - ➔ may point to invalid address !!!
  - When used, we need a “*“ to actually read/write the location they point to
  - The address the pointer itself is stored in can be determined

# Pointers and References

- References are basically pointers, but
  - Must be initialized, i.e. cannot be NULL
  - Once initialized, it is impossible to make a reference refer to another variable (i.e. address)
    - ➔ no pointer arithmetics
  - There is no "official" way to determine the address of the memory cell, the reference itself is stored in
  - When used, look just like "normal" variables syntactically

# Pointers and References

```
int i=123, j=456;

// use “*” for pointer declaration
int *ptr_to_i = NULL;
int *ptr2_to_i = &i; // assign i's address to pointer

// pointers may be uninitialized, but that’s bad practice
int *ptr3_to_i;

// use “&” for reference declaration
// NOTE: a reference MUST be initialized!
int &ref_to_i = i;

ptr_to_i = &i; // make “ptr_to_i” point to “i”

ptr3_to_i = ptr_to_i; // note that now we don't need the "&“

// “ptr_to_i” itself is stored at:
cout << “ptr_to_i’s address ” << &ptr_to_i << endl;
```


memory
address
0x0012FF60
0x0012FF64
0x0012FF68
0x0012FF6C
0x0012FF70
0x0012FF74
...
j 456
ptr_to_i 0x0012FF74
ptr3_to_i 0x0012FF74
ptr2_to_i 0x0012FF74
“ref_to_i“ 0x0012FF74
i 123
...
4 Bytes

# User Defined Data Types

- Enumerations (`enum`)
- Arrays
- Structures (`struct`)
- Classes (`class`)

# User Defined Data Types - Enumerations

- Datatype, where accepted values/elements are enumerated explicitly
  - `enum typename {element1, element2, ...};`
- Similar to a (multi-)set
- Internally elements represented by integer constants
  - Uninitialized elements are assigned the predecessor‘s value +1
  - If the first element is not initialized, it is implicitly set to 0
  - It is possible to specify the associated integer constant to each element explicitly
  - Two elements may be set to the same constant!

# User Defined Data Types - Enumerations

```
enum animation_state {   Idle,
                         Walking=3,
                         Running=3,
                         Attacking,
                         Dead };

animation_state enemy_state = Idle;

// = 0
cout << "Idle was assigned " << Idle << endl;

// = 4
cout << "Attacking was assigned " << Attacking << endl;

// ... enemy changes state ...

if(enemy_state == Dead) { /* spawn new emeny */ }
```

# User Defined Data Types - Arrays

- Datatype that stores many elements of the same type

```
element_type array_name[element_cnt];
```

- Arrays **don‘t check** if the supplied index is valid
- Arrays **don‘t store** their size (element_cnt)
- Array name is also pointer to first element

```
char c_arr[] = {'h', 'i', '\0'};
char *str = "hi";

cout << "c_arr = " << c_arr << endl;
cout << "str = " << str << endl;
```

# User Defined Data Types - Arrays

```
unsigned int idx;
const size_t arr_size=4;

float f_arr[arr_size];

// f_arr is the same as &(f_arr[0]), i.e.
// f_arr points to the array’s first element

f_arr[0] = 100.0f;
f_arr[1] = 101.0f;

// pointer arithmetics; step-size in byte = sizeof(float)
// let’s assume that sizeof(float) equals 4
// f_arr+2 increments the address f_arr by 8 bytes, which is
// exactly the size of 2 float array elements
*(f_arr+2) = 102.0f;

f_arr[3] = *(f_arr+1) + 2.0f; // = 103.0f
```

# User Defined Data Types - Structures

- Datatype that stores many elements of possibly different types

```
struct struct_name {type1 el1; type2 el2; ...};
```

- In C++ same as class with public elements only

```
struct coord {
  float x, y;
};

coord origin;

origin.x = 0.0f;
origin.y = 0.0f;
```

# Operators

- Indirection (dereference)
  - **`*var`**
- Address-of (reference)
  - **`&var`**
- Member
  - `complex_var.member`
- Member by pointer
  - `complex_var->member`
- Scope resolution
  - `scope_name::element`
- `And many more...`

# Operators

```
int var = 3;

int *ptr = NULL;


// Address-of (reference)
ptr = &var;
// => make "ptr" point to "var"


// Indirection (dereference)
int tmp = *ptr;
// => access var’s value through the
//    pointer “ptr”, which points to it
```

# Operators

```
struct coord2D {
  float x, y;
};

coord2D someCoord = {3.0f, -4.0f};//declaration and initialization
coord2D *someCoord_ptr = &someCoord;

// Member
cout << "someCoord.y = " << someCoord.y << endl;

// Member by pointer
cout << "someCoord_ptr->y = " << someCoord_ptr->y << endl;
cout << "(*someCoord_ptr).y = " << (*someCoord_ptr).y << endl;
```

# Operators

```
int var = 1; // global (file) scope

namespace scope_1 {
  int var = 2; // specific namespace
} // no ";" necessary

void foo(void) {
  int var = 3; // (function-) local scope

  // Scope resolution
  // local scope => prints "3"
  cout << "var = " << var << endl;
  // global scope => prints "1"
  cout << "::var = " << ::var << endl;
  // named scope => prints "2"
  cout << "scope_1::var = " << scope_1::var << endl;
}
```

# Gobal Scope

- Unlike Java, C++ allows to define...
  - Global functions
  - Global variables
- e.g. main – method

- Avoid using global variables
- Stick to OO design

# Gobal Scope

- Beware, Java-terminology and C++-terminology for “global scope” differ
- **Global scope** or **global namespace scope** is outermost namespace scope of a program, in which
  - *objects*
  - *functions*
  - *types* and
  - *templates* can be defined

# Gobal Scope

- A name has global namespace scope, if identifier's declaration appears outside of all
  - *blocks*
  - *namespaces*, and
  - *classes*

# Global Scope

```
int g_GlobalFoo = 0;

int multiply(int one, int two) {
    return one*two;
}

void printGlobalFoo() {
    std::cout << "Global Foo: " << g_GlobalFoo << std::endl;
}

int main(int argc, char* argv[]) {

    int foo = 2;
    int baz = 4;

    g_GlobalFoo = multiply(foo, baz);

    //will print "8"
    printGlobalFoo();

    return EXIT_SUCCESS;
}
```

# Const Correctness

```
//A non-mutable string
const string fooA("Can't be modified.");

//A pointer to a non-mutable string
const string * fooPtrA = &fooA;

//A non-mutable pointer to a non-mutable string
const string * const fooPtrB = &fooA;

//Same as fooPtrB
string const * const fooPtrC = &fooA;

//A reference to a non-mutable string
const string& fooRef = fooA;
```

# Const Correctness

- const prevents variables/objects from being modified
- Use it to…
  - …let the compiler tell you when you try to modify something that you shouldn't
  - …let the compiler optimize your code under the hood
  - …have other people understand your code better

# Passing Variables

```
//pass by value
void someFunctionA(string baz) {
     //baz holds a copy of foo
     //This assignment has NO effect on foo
     baz = "Modified by someFunctionA.";
}

...
string foo("Original Value");

someFunctionA(foo);
//Output: "Original Value"
cout << foo << std::endl;
...
```

# Passing Variables

```
//pass by reference
void someFunctionB(string& baz) {
	//function has read/write access to foo
	baz = "Modified by someFunctionB.";
}


...

someFunctionB(foo);
//Output: "Modified by someFunctionB"
cout << foo << std::endl;
...
```

# Passing Variables

```
//pass by ref-to-const
void someFunctionF(const string& baz) {
      //function has read-only access to baz
      //NOTE: baz CANNOT be NULL
      cout << "someFunctionD reads: " << baz << endl;

      //This wouldn't compile:
      //baz = "some other value";
}

...
string foo("Original Value");
//foo won't get copied
someFunctionF(foo);
...
```

# Passing Variables

```
//pass by pointer-to-const
void someFunctionD(const string* bazPtr) {

       //function has read-only access to foo
       //by dereferencing bazPtr, !! -> bazPtr can be NULL
       if (bazPtr != NULL)
              cout << "someFunctionC reads: " << *bazPtr << endl;

       //This wouldn't compile:
       //(*bazPtr) = "Modified by someFunctionC";

       //Again, changing bazPtr, which is
       //a local variable of type const string*
       //has no effect on foo
       bazPtr = NULL;
}
...
string foo("Original Value");
someFunctionD(&foo);
...
```

# Passing Variables

- Use pass-by-value for small integral types
- Use pass-by-ref for modifying (multiple) objects
- Use pass-by-const-ref for passing read-only objects
- Pass pointers if you want to check for NULL
  - But then check for NULL-ptr, really!

# Stack and Heap Memory

```
...
//A float on stack memory
float stackFloat = 1.0f;

//A float on heap memory
float* heapFloat = new float(1.0f);

//A float array on the stack (size known at compile time)
float stackFloatArray[4];
... //initialize the array!

//A float array on the heap (size can be determined at runtime)
float* heapFloatArray = new float[dynamicArraySize];
... //initialize the array!

//Be a good citizen and cleanup!
delete heapFloat;
delete[] heapFloatArray; //-> don't forget '[]' for arrays!
...
```

# Stack and Heap Memory

- Stack Memory
  - Object size known at compile-time
  - Memory of the current *frame*
  - Objects are destroyed/cleaned up when leaving current frame
  - Very fast
  - Stack Memory is limited, not suited for big arrays (→ use heap memory)

# Stack and Heap Memory

- Heap Memory
  - Object size can be determined during run-time
  - new/delete to (de)allocate single objects
  - new[]/delete[] to (de)allocate arrays
    - Initialize arrays right after allocation
  - Prefer new/delete over C malloc/dealloc for type safety
  - Write cleanup code RIGHT AFTER you wrote allocation code, really!
  - Consider using smart pointers (next session)

# Memory Leaks

```
//Create something on the heap
float* heapFloat = new float(6.0f);

//This creates a memory leak!!!
heapFloat = new float(7.0f);

//We are only cleaning up the last allocation
delete heapFloat;
```

- Each new needs its own delete!

# Common Memory Pitfalls

- Memory Leaks
  - Use tools to check for memory leaks
    - valgrind, gdb, Visual Studio, …
    - → See thread in forum for instructions
- Double Deletion
- Accessing deallocated resources
- Deletion of NULL pointers
- Underallocating arrays (out of bounds access)

# Class Declaration

```
class Person {

    public:

        Person(const string& name); //Constructor
        ~Person(); //Destructor

        const string& getName() const; //Getter
        void setName(const std::string& name); //Setter

        //Returns a string with some message
        string saySomething() const;

    private:

        string _name;

}; //<- don't forget the ";" !!!
```

# Class Declaration

- Declare classes in header files
- If you have cyclic header dependencies (A.h requires B.h and vice-versa)
  - Rethink your design
  - Use forward declarations in header and defer #include to implementation
  - Use #include as late as possible

# Constructor

```
Person::Person(const string& name) : _name(name) {
      //good place to allocate stuff (new)
}
```

- Use initializer list to initialize members
  - Required for const members or reference members
  - Initializer list has to init the members in the same order they have been declared
- Once the constructor body has finished the object is *alive* → the *destructor* is guaranteed to be called on object deallocation

# Constructor cont'd

- Multiple constructors:
  - Do not implement one in terms of the other:

```
Person::Person(const string& name) : _name(name) {}

Person::Person() {
      //This code creates only a temporary
      Person("default-name");
}
```

  - Correct:

```
Person::Person() : _name("default-name") {
      //Default Constructor
}
```

# Destructor

```
Person::~Person() {
	//deallocate members (delete,...)
}
```

- If (and only if) object is alive, destructor is **guaranteed** to be called
  - Deallocate members in here

# Const methods

- Prevents the method from modifying class-members

```
class Person {
      public:
            ...
            const string& getName() const;
            ...
};
```

- Use const methods to complete const correctness, or this:

```
void printName(const Person& p) {
      cout << p.getName() << endl;
}
```

…wouldn’t compile

# Class Inheritance

```
class Student : public Person {
        ...
};
```

- Use public inheritance to model “is-a” relations
  - Student is a Person
  - Inherited class has access to public and protected members and methods
- There are also protected and private inheritance techniques
  - ... these are more exotic, don’t bother

# Class Inheritance - virtual

```
class Person {

        public:
                Person(const string& name);
                virtual ~Person();
                ...
                virtual string saySomething() const;
                ...
};

class Student : public Person {

        public:
                Student(const std::string& name, long matNumber);
                virtual ~Student();

                long getMatNumber() const;

                virtual string saySomething() const;

        private:
                const long _matNumber;
};
```

# Class Inheritance - VTable

- Declaring a method virtual
  - Allows subclasses to *override* methods (vs. *hiding* methods)

```
Student student("Jo", 1234567);
Person& personRef = student;
//This will call Student::saySomething()
cout << personRef.saySomething() << endl;
```

  - Without declaring saySomething as virtual, the code would have called Person::saySomething instead

# Class Inheritance - VTable

- Watch out:
  - You have to use pointers or references for polymorphism:

```
Student s("Markus", 1234567);
//This will discard any relation to Student
Person p = s;
```

  - Correct:

```
Student s("Markus", 1234567);
Person& p = s;
```

# Class Inheritance – Constructor Chain

```
Student::Student(const std::string& name, long matNumber)
        : Person(name), _matNumber(matNumber)
{
}
```

- Use initializer list to initialize base class
  - There is no super() as it is in Java
- Order of creation (apply recursively):
  1. Base class, if any
  2. Data members in the order they have been declared (non-static)
  3. Constructor body is executed

# Class Inheritance – Destructor Chain

- Destructor Chain is implicit
  - You don’t have to call base classes’ destructor explicitly
  - BUT you have to declare them virtual, or...

```
Person* somePerson = new Student("Mic", 2345678);
delete somePerson;
```

  - ... Student::~Student() won’t ever be executed

# Class Inheritance - virtual

```
class Person {

        public:
              Person(const string& name);
              virtual ~Person();
              ...
              virtual string saySomething() const;
              ...
};

class Student : public Person {

        public:
              Student(const std::string& name, long matNumber);
              virtual ~Student();

              long getMatNumber() const;

              virtual string saySomething() const;

        private:
              const long _matNumber;
};
```

# Class Inheritance – Accessing base class

- To access base class methods in overriden methods:

```
string Student::saySomething() const {

    string result = Person::saySomething() \
                  + " and I am a student (nr.: " \
                  + LongToString(_matNumber) + ").";

    return result;
}
```

# Literature

- Thinking in C++ (B. Eckel)
  - Introductory, free
  - Available online:
    http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

- Accelerated Cpp (A. Koenig)
  - Introductory
    http://www.acceleratedcpp.com/

- Effective C++ (S. Meyers)
  - Essential literature for advanced C++

# That's it, folks!

- Go home, compile code...
  - Code listing are online as complete examples

- Next session:
  - More C++ features
  - Standard template library
  - A few How To's
  - Your questions → forum!

- Questions?