University of Duisburg-Essen
Networked Embedded Systems Group
Dept. of Computer Science & Business Information Systems
Schützenbahn 70
45127 Essen, Germany

Programming in C/C++
3 – Introduction to C++

Prof. Dr. Pedro José Marrón

Overview

- Function overloading
- Namespaces
- Simple output (“preview” for example source and exercises)
- Some new features in C++11
  - Automatic type inference
  - Range-based for loops
  - Lambda functions
- C++ incompatibilities with C (that are relevant in practice)
- Casting
- Enums (also C++11 strongly typed enums)
- Operator keywords

Function Overloading

- Function overloading
  - Functions in same scope with same name and different parameters
  - Should perform similar tasks
    - I.e., function to square **`int`**s and function to square **`float`**s

```
int square(int x) { return x * x; }
float square(float x) { return x * x; }
```

- Overloaded functions distinguished by signature
  - Based on name and parameter types (order matters)
    - Not on return type
  - Name mangling
    - Encodes function identifier with parameters
  - Type-safe linkage
    - Ensures proper overloaded function called

Namespaces

- Program has identifiers in different scopes
  - Sometimes scopes overlap, lead to problems
- Namespace defines scope
  - Place identifiers and variables within namespace
  - Access with ***namespace_name*::*member***
  - Not guaranteed to be unique
  - No relationship with file location
  - Unnamed namespaces are global
    - Need no qualification
    - Preferred Alternative for “static” functions
  - Namespaces can be nested

```
void func() {}
namespace {
// corresponds to static … in C
   void f() {
      …
   }
}

namespace outer {
   namespace inner {
      void func() {}
      void f() {
         // force access to global
         // namespace
         ::func();
      }
   } // namespace inner
} // namespace outer

outer::inner::f();
```

Namespaces (2)

- **`using`** statement
  - **`using namespace`** ***`namespace_name;`***
  - Members of that namespace can be used without preceding ***`namespace_name`*****`::`**
    - Can also be used with individual member
  - Examples
    - **`using namespace std`**
      - Discouraged by some programmers, because includes entire contents of **`std`**
    - **`using namespace std::cout`**
      - Can write **`cout`** instead of **`std::cout`**
- NEVER use in header files

Simple Output

- std::cout in iostream
- Output with <<
  - Typesafe
  - Cascading
- std::endl
  - Endline
  - Flushes buffer
- For example sources
  - Assume “using std::cout;”
- Details in dedicated chapter

```
#include <iostream>

int i = 13;

std::cout << "12" << 3 << "\n";
std::cout << i << std::endl;

using std::cout;
int* pi = &i;
cout << "abc\n" << pi << "\n";
```

Automatic Type Inference (in C++11)

- Compiler can infer automatically the type of a variable with explicit initialization

```
int a = 1;
auto int_variable = a;
auto also_an_int = 2;
```

- This eases programming a lot when using STL iterators (see later)
  Instead: `map<string,string>::iterator itr = m.begin();`
  write: `auto itr = m.begin();`
- Keyword „auto“ is a storage-class specifier in C!

- BUT: Do not use it in the exercises since we want to see that you understand what you are doing!

Range-based for Loops (in C++11)

- Syntax of `for` statement now allows iteration over range of elements as know from java!

```
int arr[] = {1,2,3,4,5};
for (int& x : arr)
   x++;
```

  - This increases the value of each array element since we use a reference to each element!
- Great when combined with „auto“:

```
for (auto& x : arr)
```

- Works for normal arrays, initializer lists (see later) and any type that has begin() and end() functions returning iterators (see also later)

Lambda-Functions (in C++11)

- Motivation
  - For some functions, e.g. qsort, simple action functions need to be passed as parameter

```
int arr[] = {9,3,6,8,4,1,5};

int compareint(const void *a, const void *b) {
  return *(int*)a - *(int*)b;
}

qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), &compareint);
```

  - Action functions have no other purpose and are mostly used only once
  - But introduce a new name

Lambda-Functions (2)

- C++11 supports anonymous functions (called lambda functions)
- Syntax:

```
[ optional_capture_list ]
( parameter_declaration )
-> return_type
{ statements }
```

  - „[…]“ tells the compiler that it should create a lambda function
  - „(parameter_declaration)“ is optional, then „()“ is assumed
  - „-> return_type“ is optional, then deduced by the compiler
    - If statements end with „return“ it‘s the type of the returned expression
    - „void“ otherwise

Lambda-Functions (3)

- Example: Hello world with lambda function

```
int main() {
  auto func = [] { std::cout << "Hello world"; };
  func(); // call the function
  // or in short:
  [] { std::cout << "Hello world"; } ();
}
```

- Example: Remember the traditional qsort call

```
int compareint(const void *a, const void *b) {
  return *(int*)a - *(int*)b;
}

qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), &compareint);
```

  - Now with lambda function

```
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int),
      [](const void *a, const void *b) { return *(int*)a - *(int*)b; }
     );
```

Lambda-Functions (4)

- Imagine that you don‘t want to sort a list, but only the indices to this list (also using other C++ features like strings):

```
#include <iostream>
#include <string>
#include <algorithm>

int main(void) {
  std::string names[] = {"Tom", "Alice", "Jane", "Bob"};
  int indices[] = {0,1,2,3};

  std::sort(indices, indices + sizeof(indices)/sizeof(int),
            [&](int a, int b) { return names[a] < names[b]; }
           );

  for (int i = 0; i < sizeof(indices)/sizeof(int); i++)
    std::cout << names[indices[i]] << std::endl;
}
```

  - „[&]“ gives lambda function access to names[]. Without it, names[] has to be global!

Lambda-Functions (5)

- Capture list („[…]“) specifies, how and which local variables of the scope, in which the lambda function is defined, can be accessed inside the lambda function
  - [] : no variables
  - [x, &y] : x captured by value, y captured by reference
  - [&] : capture all used variables by reference
  - [=] : capture all used variables by making a copy
  - [&, x, z] : capture all used variables by reference, except for x and z which is copied
  - [=, &y] : capture all used variables by copying them, but create a reference for y
  - [this]: capture the pointer of the enclosing class

Rvalue References (in C11++)

- lvalue: may also appear on left side of an assignment
  - Basically an expression referring to a memory location
- rvalue: can only be used on right side of an assignment
- Non-const references can only bind to lvalues

```
int f() { return 1; }
int i;
int& ri = i;   // OK
int& rf = f(); // error: cannot init non-const int& with int rvalue
ri = 8;        // sets i to 8
```

- Const references can bind to lvalues and rvalues
  - But values cannot be changed

```
const int& rcf = f(); // This is ok!
rcf = 15;             // this is forbidden since reference is const
```

Rvalue References (2)

- In C++11, references to rvalues are introduced
  - Denoted by &&
  - Can **only** bind to rvalues, not lvalues!

```
int f() { return 1; }
int&& rrf = f();  // rvalue reference!
rrf = 24;         // OK!

int i;
int&& rrf2 = i;   // error! cannot bind 'int' lvalue to 'int&&'

int&& rrf3 = 12;  // OK!
rrf3 += 30;       // OK!
```

- Temporary stores the value, which is accessed by rvalue ref
- Can appear on left side of assignments!

Rvalue References (3)

- Why is this needed? We have pointers!
- C++ focuses very much on references (see next chapters)
  - Safer to use than pointers
  - OO features like automatic call of constructors and destructors or operators do not work with pointers
- Problem: Lot of data might be copied in constructors/assignments
- Example: Return value of f() is copied in assignment
  - With rvalue reference, temporary returned by f() is used further

```
struct X { /* something large */ }

X f() { ... return a_new_X; }

X myX = f();     // the complete struct is copied to myX
X&& rrX = f();   // no copy! rrX holds reference to temporary
```

Rvalue References (4)

- With function overloading, functions can behave differently for lvalue and rvalue parameters
  - Will become important with classes – see next chapter

```
string getName();

void printIt (const string& str) {
   cout << str;
}

void printIt (string&& str) {
   cout << str;
}

string s("c is great");
printIt(s);          // 1st printIt since s is lvalue

printIt(getName()); // 2nd printIt since getName returns mutable rvalue.
                    // takes precedence over 1st printIt which would
                    // also accept a rvalue (since param is const)!
```

C++ 14 features

- C++ 14 is a small extension to C++ 11. Below are some of the new features introduced:
  - Deduction of return type for all functions
    - This was possible in C++11 for Lambdas, but now available to all
    - Functions should be declared with `auto` as the return type
    - E.g. `auto deduceReturnType(std::string input);`
  - Template variables (not only for functions/classes)
    - Actual type depends on how the variable is read
  - Binary literals with `0b` or `0B` prefix
    - E.g. `auto binary_literal = 0b11010011;`

C++14 features (2)

- Generic lambda parameters
  - In C++11, lambda parameters need to be concrete types
  - In C++14, they can be declared with `auto`
  - E.g. `auto lambda = [](auto x) { return x*x;};`

- Deprecated attribute
  - Mark entity as deprecated; its use is discouraged
  - E.g.: `[[deprecated]] int f();`

- Digit separators (for easier human reading)
  - single character quote as separators in numeric literals
  - E.g. `auto int_literal = 1'000'000;`

C vs. C++

- Additional keywords cannot be used as identifiers
  - Duh
  - Including C, C++ and C++0x (the next standard version):
  - and, default, noexcept, template, and_eq, delete, not, this, alignof, double, not_eq, thread_local, asm, dynamic_cast, nullptr, throw, auto, else, operator, true, bitand, enum, or, try, bitor, explicit, or_eq, typedef, bool, export, private, typeid, break, extern, protected, typename, case, false, public, union, catch, float, register, using, char, for, reinterpret_cast, unsigned, char16_t, friend, return, void, char32_t, goto, short, wchar_t, class, if, signed, virtual, compl, inline, sizeof, volatile, const, int, static, while, constexpr, long, static_assert, xor, const_cast, mutable, static_cast, xor_eq, continue, namespace, struct, decltype, new, switch

C vs. C++ (2)

- C: void* automatically cast to all pointer types
  - C++ explicit cast necessary
  - Other direction still automatic
- C: const int* automatically cast to int*
- C++: struct, enum, union implies typedef
  - typedef struct A {} A; // not valid in C++
- ‘a’: C signed int; C++ char ((un)signed implementation defined)

C vs. C++ (3)

- Differences in linking requires extern “c” for pure C functions in header files

- Preferred naming for standard include files
  - Omit .h
    - #include “iostream”;
  - C standard library headers for C++
    - #include “cstdio”;
  - Old compilers may not support this

```
// Header file
#ifdef __cplusplus
// If C++ compiler, use C linkage
extern "C" {
#endif

// functions have C linkage
void foo();

struct bar { /* ... */ };

// C++ compiler -> end C linkage
#ifdef __cplusplus
}
#endif
```

Casting

- C style cast valid but deprecated
- “Normal casts”, e.g. double to int or Derived* to Base*
  - Valid only if possible, expresses “intent”
  - static_cast<type>(expression)
- Reinterpret bit pattern
  - Not for normal casts
  - “Accepted” only for casting pointers
  - reinterpret_cast<type>(expression)
- Explained later:
  - dynamic_cast: polymorphism; safe cast base pointer to child
  - const_cast: remove constness

Enums in C++

- In C no type safety for enums (see Chapter 2)
- In C++, no implicit conversion of integer or value of one enum type to another enum type

```
enum apple_t {ELSTAR, BOSKOOP, GRANNY_SMITH};
enum orange_t {VALENCIA, NAVEL, BLOOD};
orange_t myO = BOSKOOP; // error: cannot convert apple_t to orange_t
myO = 2; // error: invalid conversion from int to orange_t
```

- However, comparisons and calculations using values of different enums are still possible

```
apple_t myA = BOSKOOP;
orange_t myO = BLOOD;
bool comp = myO > myA; // prints just a warning
comp = GRANNY_SMITH > VALENCIA; // prints just a warning
int somejuice = (ELSTAR - BOSKOOP) / NAVEL; // perfectly fine!
```

- As in C names of members in different enums have to be unique

C++11: Strongly typed enums

- Type-safe enums that are not implicitly converted to integers
  - Cannot be compared to integers or to enums of different types
- Values belong to the enum class
  - Same names can appear in different enums
  - But scope of value has to be given using ::

```
enum class apple_t {ELSTAR, BOSKOOP, GRANNY_SMITH};
enum class orange_t {VALENCIA, NAVEL, BLOOD};
enum class cities_t {PARIS, VALENCIA, ROME}; // fine! No name clash

apple_t myA = apple_t::BOSKOOP; // access needs ::
orange_t myO = orange_t::BLOOD;

int i = myA; // error: cannot convert apple_t to int
bool comp = myO > myA;           // error: no > operator and no conversion
comp = GRANNY_SMITH > VALENCIA; // apple_t <-> orange_t defined};
```

Operator Keywords

- Operator keywords
  - Can be used instead of operators
  - Useful for keyboards without `^` `|` `&` etc.
- Very rarely used
  - Be aware but do not use them
- Exist as macros in C (iso646.h)

Operator Keywords

Operator	Operator keyword	Description
Logical operator keywords		
&&	and	logical AND
||	or	logical OR
!	not	logical NOT
Inequality operator keyword		
!=	not_eq	Inequality
Bitwise operator keywords		
&	bitand	Bitwise AND
|	bitor	Bitwise inclusive OR
^	xor	Bitwise exclusive OR
~	compl	Bitwise complement
Bitwise assignment operator keywords		
&=	and_eq	Bitwise AND assignment
|=	or_eq	Bitwise inclusive OR assignment
^=	xor_eq	Bitwise exclusive OR assignment

boost C++ Libraries

- http://www.boost.org/
- Peer-reviewed portable C++ libraries
  - Very good quality
- Many contributors involved in C++ standardization
  - Some libraries got accepted for C++11 standard
- Collection of libraries, not one homogeneous library
- Good design – not easy to read
- We make some references to convenient extensions to the C++ standard library

Links

- For an overview of the new C++11, C++14 and C++17 features see https://isocpp.org/blog/2017/07/cpp17-14-11-a-cheatsheet-of-modern-c-language-and-library-features-anthony