Advanced Systems Programming

Introduction to C++

Martin Küttler

September 19, 2017
About this presentation

► This presentation is not about learning programming or every C++ feature.
► It is a short introduction to modern C++.
► I focus on interactive coding, and exercises.
► Please, do ask questions!
About this presentation

- This presentation is not about learning programming or every C++ feature.
- It is a short introduction to modern C++.
- I focus on interactive coding, and exercises.
- Please, do ask questions!

Slides and Code:

https://os.inf.tu-dresden.de/Studium/SysProg/

svn co http://svn.inf.tu-dresden.de/repos/advsysprog/day2
About this presentation

- This presentation is not about learning programming or every C++ feature.
- It is a short introduction to modern C++.
- I focus on interactive coding, and exercises.
- Please, do ask questions!

Slides and Code:

https://os.inf.tu-dresden.de/Studium/SysProg/

svn co http://svn.inf.tu-dresden.de/repos/advsysprog/day2

More information:

(e.g.) A Tour of C++, https://isocpp.orgs/tour
C++

- Programming language, mostly superset of C
- Supports object-oriented programming and generic programming
- Aims to provide zero-overhead abstractions
- Standardized by ISO
Fundamental Types

- **bool**
- **Integers**
  - char, short (int), int, long (int),
    long long (int)
  - Each can be unsigned or signed (the default for most)
- **Floating point values**
  - float, double, long double
- **void**
  - indicates no type
- **auto**
  - Not a type by itself, can deduce the correct (tm) type.
Composed types

- Pointers: $T \cdot p$, Address of variable: $&\text{var}$
Composed types

- Pointers: T* p, Address of variable: &var
- (L-value) References: T& r
Composed types

- Pointers: \( T* p \), Address of variable: \&var
- (L-value) References: \( T& r \)
- R-value References: \( T&& r \) (more on that later)
Composed types

- Pointers: $T \ast \ p$, Address of variable: $\& \ var$
- (L-value) References: $T \& \ r$
- R-value References: $T \& \& \ r$ (more on that later)
- Arrays: $T \ array \ [N]$
Composed types

- **Pointers:** \( T* \ p \), Address of variable: \&\ vari
- **(L-value) References:** \( T& \ r \)
- **R-value References:** \( T&& \ r \) (more on that later)
- **Arrays:** \( T \ array[N] \)
- **Functions:** \( T \ fun(A1, \ldots, \ An) \)
Composed types – Examples

- `int a[4] = {1, 1, 2, 3};`
  ```
  1 1 2 3
  ```
Composed types – Examples

- int a[4] = {1, 1, 2, 3};
  
  | 1 | 1 | 2 | 3 |

- int a[2][3] = {{5, 8, 13}, {21, 34, 55}};
  
  | 5 | 8 | 13 | 21 | 34 | 55 |
Composed types – Examples

- \( \text{int } a[4] = \{1, 1, 2, 3\}; \)
  
  \[
  \begin{array}{cccc}
  1 & 1 & 2 & 3 \\
  \end{array}
  \]

- \( \text{int } a[2][3] = \{{5, 8, 13}, \{21, 34, 55}\}; \)
  
  \[
  \begin{array}{cccc}
  5 & 8 & 13 & 21 & 34 & 55 \\
  \end{array}
  \]

- \( \text{char } str[10] = "Test\[1\]; \)
  
  \[
  \begin{array}{cccc}
  T & e & s & t \[ & 1 & \backslash 0 & ? & ? & ? \\
  \end{array}
  \]
Composed types – Examples

- `int a[4] = {1, 1, 2, 3};`
  - 1 1 2 3

- `int a[2][3] = {{5, 8, 13}, {21, 34, 55}};`
  - 5 8 13 21 34 55

- `char str[10] = "Test\n1";`
  - Test 1 \0 ?? ??

- `char a[3][6] = {"One", "Two", "Three"};`
  - One \0 ?? ?? Two \0 ??
    - Three \0
Composed types – Combinations

▶ int increase(int a, int step=1);
Composed types – Combinations

- int increase(int a, int step=1);
- int matrix[3][4];
Composed types – Combinations

- `int increase(int a, int step=1);`
- `int matrix[3][4];`
- `int* p = &matrix[2][0];`
Composed types – Combinations

- \texttt{int increase(int a, int step=1);}
- \texttt{int matrix[3][4];}
- \texttt{int* p = &matrix[2][0];}
- \texttt{char** argv;}

Try cdecl, or cdecl.org
Composed types – Combinations

- `int increase(int a, int step=1);`
- `int matrix[3][4];`
- `int* p = &matrix[2][0];`
- `char** argv;`
- `int (**p2)[3][4]`
Composed types – Combinations

- `int increase(int a, int step=1);`
- `int matrix[3][4];`
- `int* p = &matrix[2][0];`
- `char** argv;`
- `int (*p2)[3][4] = &matrix;`
Composed types – Combinations

- `int increase(int a, int step=1);`
- `int matrix[3][4];`
- `int* p = &matrix[2][0];`
- `char** argv;`
- `int (*p2)[3][4] = &matrix;`
- `int (**a[10])(short, bool)(int*)[4];`
Composed types – Combinations

- int increase(int a, int step=1);
- int matrix[3][4];
- int* p = &matrix[2][0];
- char** argv;
- int (*p2)[3][4] = &matrix;
- int (**a[10])(short, bool)(int*)[4];
- Try cdecl, or cdecl.org
Use `const` to indicate immutable values

- `const int a;` // or `int const a;`
Use `const` to indicate immutable values

- `const int a; // or int const a;`
- `int const * p; // pointer to constant int`
Use `const` to indicate immutable values

- `const int a; // or int const a;
- `int const * p; // pointer to constant int
- `int *const p; // constant pointer to int
Use `const` to indicate immutable values

- `const int a;` // or `int const a;`
- `int const * p;` // pointer to constant int
- `int *const p;` // constant pointer to int
- `int const * const p;`  // constant pointer to constant int
**Const**

Use `const` to indicate immutable values

- `const int a;`  // or `int const a;`
- `int const * p;`  // pointer to constant int
- `int *const p;`  // constant pointer to int
- `int const * const p;`
  // constant pointer to constant int

Use `constexpr` for values that are known at compile time

- `int constexpr answer = 42;`  // ok
**Const**

Use `const` to indicate immutable values

- `const int a; // or int const a;
- `int const * p; // pointer to constant int
- `int * const p; // constant pointer to int
- `int const * const p;
  // constant pointer to constant int

Use `constexpr` for values that are known at compile time

- `int constexpr answer = 42; // ok
- `int constexpr val = foo(x);`
Use `const` to indicate immutable values

- `const int a; // or int const a;
- `int const * p; // pointer to constant int
- `int *const p; // constant pointer to int
- `int const * const p;
  // constant pointer to constant int

Use `constexpr` for values that are known at compile time

- `int constexpr answer = 42; // ok
- `int constexpr val = foo(x);
  // might be ok`
Dynamic Memory

- Manually, C-Style

```c
Big *data = (Big*)malloc(sizeof(Big));
// use data ...
free(data);
```
Dynamic Memory

- Manually, C-Style
  
  ```c
  Big *data = (Big*)malloc(sizeof(Big));
  // use data ... 
  free(data);
  ```

- Manually, C++-Style
  
  ```cpp
  Big *data = new Big;
  // use data ... 
  delete data;
  ```

- Automatic, C++
  
  ```cpp
  auto data = make_unique<Big>();
  // just use data, no need to free
  // or, when you keep multiple references
  auto data = make_shared<Big>();
  ```

When a container class (e.g. string, vector, etc.) makes sense, use it!
Dynamic Memory

- Manually, C-Style

```c
Big *data = (Big*)malloc(sizeof(Big));
// use data ...
free(data);
```

- Manually, C++-Style

```cpp
Big *data = new Big;
// use data ...
delete data;
```

- Automatic, C++

```cpp
auto data = make_unique<Big>();
// just use data, no need to free
// or, when you keep multiple references
auto data = make_shared<Big>();
```
Dynamic Memory

- Manually, C-Style
  ```
  Big *data = (Big*)malloc(sizeof(Big));
  // use data ...
  free(data);
  ```

- Manually, C++-Style
  ```
  Big *data = new Big;
  // use data ...
  delete data;
  ```

- Automatic, C++
  ```
  auto data = make_unique<Big>();
  // just use data, no need to free
  // or, when you keep multiple references
  auto data = make_shared<Big>();
  ```

- When a container class (e.g. string, vector, etc.) makes sense, use it!
Argument Passing

- “Call by Value”: `void foo(T a) {}`
  Argument gets copied, good for cheap-to-copy types.

- “Call by Reference”: `void foo (T* a)` or `void foo(T& a)`
  Only reference is passed, no values are copied. Original values can be changed in the function.
Argument Passing

- "Call by Value": `void foo(T a) {}`
  Argument gets copied, good for cheap-to-copy types.

- "Call by Reference": `{}`
  `void foo(T* a)` or
  `void foo(T& a)`
  Only reference is passed, no values are copied. Original values can be changed in the function.

- Variant: `{}`
  `void foo(T const& a)`
  Argument is not copied, but cannot be changed.
Argument Passing

- "Call by Value": `void foo(T a) {}`
  Argument gets copied, good for cheap-to-copy types.
- "Call by Reference": `void foo (T* a)` or `void foo(T& a)`
  Only reference is passed, no values are copied. Original values can be changed in the function.
- Variant: `void foo(T const& a)`
  Argument is not copied, but cannot be changed.
- If ownership should be transferred, pass `unique_ptr<T>` or `shared_ptr<T>`. 

Exercise 1

Write a function `split` that takes a string `s` and an optional `char c` (default value `',`) and returns a `std::vector<std::string>` that contains the substrings of `s` that lie between successive occurrences of `c` in `s`.

Tips:

- `#include <vector>` and `#include <string>`
- For a `std::string s`, `s.begin()` and `s.end()` give iterators to begin and end.
- Alternatively, `s[i]` works just like for arrays (c-strings).
- When in doubt: Ask me or the internet.
Abstract Data types

Data type plus functions

- Couple data and access functions.
- Provide some access control.
- Maintain invariants, enforce initialization.
Syntax

class Name {
    [private:]
        [members]
    [protected:]
        [members]
    [public:]
        [members]
};
Constructor and Destructor

- **Constructor:** Member function with same name as class; automatically called when an object is created.
- **Destructor:** Member function with name `~classname`; automatically called when object is destroyed.
Overloading

- Functions (including member functions, and constructors) can be overloaded.
- Operators are also functions, with the name `operator OP`

Special constructor overloads:
- “copy constructor”: `Big::Big(Big const& other)`; Used when constructing a copy of an existing `Big` object.
- “move constructor”: `Big::Big(Big && other)`; Used when constructing a new `Big` object from an existing one, when data should be moved from the old object, leaving it ’empty’ afterwards.
Inheritance

You can create an enhanced/specialized version of a class by inheriting from the base class.

```cpp
class Derived : public Base
{
    // ...
};
```

The access specifier before `Base` further restricts access to the members of the base class. Mostly just `public` is used.
Virtual functions

Member functions can be declared

- **virtual**: This allows the Derived class to override the method with its own definition.

- **pure virtual** (e.g. `virtual void foo() = 0`): The function is virtual and has no definition. The corresponding class is an abstract class then.
Exercise 2

Implement a linked list of paths.

- Fill in the missing parts in `linked-list.h`
- Implement the functions in `linked-list.cc`
- Test with `test-list`. 