Advanced Systems Programming

Introduction to C++

Martin Küttler

September 23, 2016
About this presentation
About this presentation

This presentation is not about

▶ learning to program
About this presentation

This presentation is not about

- learning to program,
- learning every C++ feature.
About this presentation

This presentation is not about

► learning to program,
► learning every C++ feature.

I focus on

► an introduction to modern C++
About this presentation

This presentation is not about

- learning to program,
- learning every 
  C++ feature.

I focus on

- an introduction to modern 
  C++,
- interactive examples and exercises.

Slides and Code: see website

About this presentation

This presentation is not about
  ► learning to program,
  ► learning every C++ feature.
I focus on
  ► an introduction to modern C++,
  ► interactive examples and exercises.
Slides and Code:

  see website
About this presentation

This presentation is not about

► learning to program,
► learning every C++ feature.

I focus on

► an introduction to modern C++,
► interactive examples and exercises.

Slides and Code:

see website

More information:

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 type.
Composed types

- Pointers: T* p, Address of variable: &var

- References:
  - (L-value) References: T& r
  - (R-value) References: T&& r (more on that later)

- Arrays: T array[N]

- Functions: T fun(A1, ..., An)
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: \( T* \ p \), 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: \& \( \text{var} \)
- (L-value) References: \( T& \ r \)
- R-value References: \( T&& \ r \) (more on that later)
- Arrays: \( T \ \text{array} [N] \)
- Functions: \( T \ \text{fun}(A1, \ldots, An) \)
Composed types – Examples

```c
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

- `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\1";
  
  Test 1 \0 ?? ??`
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";`
  
  | T | e | s | t | \n | 1 | \0 | ? | ? | ? |

- `char a[3][6] = {"One", "Two", "Three"};`
  
  | O | n | e | \0 | ? | ? | T | w | o | \0 | ? | ? | T | h | r | e | e | \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

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

- `int` `increase(int a, int step=1);`
- `int` `matrix[3][4];`
- `int*` `p = &matrix[2][0];`
- `char**` `argv;`
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`
Use `constexpr` for values that are known at compile time

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

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`
- `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`
Compound types

- Data structures

  ```c
  struct name {
  short a; 
  char b; 
  }
  ```

- Unions

  ```c
  union name {
  short a; 
  char b; 
  }
  ```
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;
  ```

- **(Mostly) 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
  ```c++
  Big *data = new Big;
  // use data ...
  delete data;
  ```

- (Mostly) Automatic, C++
  ```c++
  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
  
  ```c
  Big *data = (Big*)malloc(sizeof(Big));
  // use data ...
  free(data);
  ```

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

- (Mostly) 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!
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[* or &] 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[* or &] 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.
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.
- **this**: implicitly declared pointer to current object in member functions.
Overloading

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

Special constructor overloads:

- “default constructor”: `Big::Big() {}`
  Can be called without an argument.
- “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

Write a subclass Subprocess of class File, which

- Starts a new process, executes a given command in it, and stores a file descriptor to read the output of that process.
- Frees the processes resources when closing the file.

You might want to use the system calls pipe, fork, dup, and exec. man is your friend.
Exercise 3a

Write a linked list of Files. Use the header linked-list.h and provide the appropriate implementations in a cc-file.
Exercise 3b

Write a `UniquePointer` to File. It should:

- (If needed) allocate and delete a Big automatically.
- Be able to be empty (hold `nullptr`)
- Not create memory leaks or double frees.
- Provide a way to retrieve ownership of the pointer, leaving an empty unique pointer.
- Provide a way to get the raw pointer, without releasing ownership.
- Implement `operator *` and `operator ->` to access the File.