Modern C++, From the Beginning to the Middle

Ansel Sermersheim & Barbara Geller
emBO++
March 2021

Introduction

- Prologue
- Where is the Beginning
- Data Types
- Pointers / References
- Value Categories
- Expressions
- Parameter Passing
- Examples

Prologue

- Credentials
  - every library and application is open source
  - our development uses cutting edge C++ technology
  - all source code hosted on github
  - prebuilt binaries are available on our download site
  - documentation is generated by DoxyPress

  - youtube channel with videos focused mostly on C++
  - frequent speakers at multiple conferences
    - CppCon, CppNow, emBO++, MeetingC++, code::dive
  - numerous presentations for C++ user groups
    - United States, Germany, Netherlands, England

Prologue

- Maintainers and Co-Founders
  - CopperSpice
    - cross platform C++ libraries
  - DoxyPress
    - documentation generator for C++ and other languages
  - CsString
    - support for UTF-8 and UTF-16, extensible to other encodings
  - CsSignal
    - thread aware signal / slot library
  - CsLibGuarded
    - library for managing access to data shared between threads

Where is the Beginning

- What can you Define?
  - what makes something a data type
  - how does an expression relate to a data type

  - is a reference an object that refers to a pointer
  - can you pass a pointer by reference
  - are pointers of value
  - do we really need references

  - are these just different words for the same thing
    - reference, lvalue, lvalue reference

Where is the Beginning

- C++11 was a New Beginning
  - defined new data types
  - new value categories
  - defined semantics
  - constexpr, lambda expressions, smart pointers
  - memory model, atomics, mutexes, threading library

- C++ standard
  - C++98 standard is 832 pages *(page size: letter, font 10 pt)*
  - C++11 standard is 1222 pages
  - C++14 standard is 1261 pages
  - C++17 standard is 1485 pages
  - C++20 standard is 1683 pages *(page size: A4, font 8 pt)*

Data Types

- Definition of a data type
  - data types are defined by two characteristics
    - set of possible values
    - operations which can be done on or with the values

Data Types

- Primitive or Simple Data Types
  - basic low level types which must be provided by the language
  - only one value is associated with a given variable

  - examples in C++
    - char, int, bool, double, float

Data Types

- Built In Data Types
  - types which are provided by the language as a convenience
  - exact types will vary depending on the programming language

  - examples in C++
    - std::array, std::complex, std::list, std::vector

Data Types

- Composite or Compound Data Types
  - derived from more than one primitive and/or built in type
  - creating a composite type produces a new data type

  - examples in C++
    - class, structure

Data Types

- User Defined Data Types
  - declared by the developer in their source code -OR-
  - user types created in a third party library

  - examples in C++
    - `enum class Spices { mint, basil, salt, pepper };`
    - `class QString;`
    - `class Employee;`

Data Types

- Abstract Data Type
  - any type which does not specify an implementation
  - definition of a Stack includes the push() and pop() functions
    - well defined in computer science
    - implementation depends on the storage container
  - an abstract class does not implement all methods it declares
  - you do not directly instantiate an abstract class
  - users should create a subclass and then instantiate the child class

Pointers

- Pointer Data Type
  - values
    - nullptr or an address where some data is located
  - operations
    - assignment
    - dereference
    - comparison
    - subtraction of two pointer values
    - addition or subtraction with an integer value

Pointers

- Pointer Data Type
  - declaring a pointer must include the type of what is being stored
  - size of the pointer type is based on the platform
  - required allocation for the data is determined by the type used in the pointer declaration

```
int * var1;          // var1 declares a ptr to a value of type int

Widget * var2;       // var2 declares a ptr to a value of type Widget
```

References

- Reference Data Type
  - values
    - determined by the values of the type being referenced
  - operations
    - determined by the operations of the type which is referenced
    - might be limited to a subset of the operations if there are qualifiers such as const

References

- Reference Data Type
  - lvalue reference
    - declared object can be modified by the called function and then observed by the original caller
  - const reference
    - called function can not modify the passed object
  - rvalue reference
    - declared object can be modified by the called function however the original caller should never observe the changes

References

- Reference Data Type
  - declaring an lvalue reference involves specifying the type and a single & before the variable

  - countB is a variable which is bound to countA
  - modifying the value of countB will change the value of countA

```
int countA = 12;
int & countB = countA;
```

References

- Reference Data Type
  - declaring a const reference involves specifying the type and a single & before the variable

  - countB is a variable which is bound to countA
  - modifying the value of countB is not permitted
  - changing countA is allowed and will be visible by countB

```
int countA = 17;
const int & countB = countA;
```

References

- Reference Data Type
  - declaring an rvalue reference involves specifying the type and a double && before the variable

  - countB is a variable which is bound to countA
  - modifying the value of countB will change the value of countA
  - after the std::move() countA should never be observed

```
int countA = 8;
int && countB = std::move(countA);
```

Value Categories

- Value Categories
  - five main groups
    - glvalue, prvalue, xvalue, lvalue, rvalue
  - every object, variable, or expression is either . . .
    - an lvalue or an rvalue
  - if any of these are true it is an lvalue
    - has an identity
    - has a name
    - resides at a memory location

Value Categories

- lvalue
  - button is an lvalue and its data type is pointer to Widget
  - *button is an lvalue and its data type is Widget

```
Widget * button = new Widget;
```

- rvalue
  - passed value is an rvalue and its data type is std::string
  - result is an lvalue and its data type is int

```
int result = someFunction( std::string(“emBO”) );
```

Expressions

- Definition of an Expression
  - every expression has two attributes
    - data type
    - value category
  - evaluation of an expression always generates a result

```
int sum;                        // line 1
sum = 10 + 20;                  // line 2


auto index = getIndex();        // line 3
++index;                        // line 4
```

Expressions

- Definition of an Expression
  - getIndex() is a function call expression
  - value category depends on the return type
    - returns by value then it is an rvalue (prvalue)
    - returns an lvalue reference then it is an lvalue
    - returns an rvalue reference then it is rvalue (xvalue)

```
auto index = getIndex();        // line 3
++index;                        // line 4
```

Parameter Passing

- Passing Arguments
  - functions in C always receive parameters by value
    - programmers call this “pass by value”
  - functions in C++ can receive parameters by value or by reference
    - programmers call this “pass by value”
    - programmers call this “pass by reference”
  - “pass by X” is misleading . . .

Parameter Passing

- Example 1
  - int data type, value is 27

```
int dayA = 27;
```

Variable	Memory Address	Value
dayA	1000	27

Parameter Passing

- Example 1.1

```
int dayA = 27;
myFunc( dayA );


void myFunc( X dayB );
```

Options for X
`const int dayB`
`int dayB`
`const int & dayB`
`int & dayB`

Parameter Passing

- Example 1.2

```
int dayA = 27;
myFunc( &dayA );


void myFunc( X dayB );
```

Options for X
`const int * dayB`
`int * dayB`
`const int * const & dayB`
`int * const & dayB`
`const int * && dayB`
`int * && dayB`

Parameter Passing

- Example 1.3

```
int dayA = 27;
myFunc( *dayA );


void myFunc( X dayB );
```

Options for X
dereference of an int data type is not valid

Parameter Passing

- Example 2
  - pointer data type, int value is 3

```
int *monthA = new int(3);
```

Variable	Memory Address	Value
monthA	1000	5000
	5000	3

Parameter Passing

- Example 2.1

```
int * monthA = new int(3);
myFunc( monthA );


void myFunc( X monthB );
```

Options for X
`const int * monthB`
`int * monthB`
`const int * const & monthB`
`int * & monthB`
`int * const & monthB`

Parameter Passing

- Example 2.2

```
int * monthA = new int(3);
myFunc( &monthA );


void myFunc( X monthB );
```

Options for X
`const int * const * monthB`
`int * const * monthB`
`int ** monthB`
`const int * const * const & monthB`
`int * const * const & monthB`
`int ** const & monthB`
`int ** && monthB`
`int * const * && monthB`

Parameter Passing

- Example 2.3

```
int * monthA = new int(3);
myFunc( *monthA );


void myFunc( X monthB );
```

Options for X
`const int monthB`
`int monthB`
`const int & monthB`
`int & monthB`

Parameter Passing

- Example 3
  - reference data type, int value is 2021

```
int yearA   = 2021;
int & yearB = yearA;
```

Variable	Memory Address	Value
yearA	1000	2021
yearB	1000	2021

Parameter Passing

- Example 3.1

```
int yearA   = 2021;
int & yearB = yearA;
myFunc( yearB );


void myFunc( X yearC );
```

Options for X
`const int yearC`
`int yearC`
`const int & yearC`
`int & yearC`

Parameter Passing

- Example 3.2

```
int yearA   = 2021;
int & yearB = yearA;
myFunc( &yearB );


void myFunc( X yearC );
```

Options for X
`const int * yearC`
`int * yearC`
`const int * const & yearC`
`int * const & yearC`
`const int * && yearC`
`int * && yearC`

Parameter Passing

- Example 3.3

```
int yearA   = 2021;
int & yearB = yearA;
myFunc( *yearB );


void myFunc( X yearC );
```

Options for X
dereference of an int data type is not valid

Parameter Passing

- Reference to a Pointer
  - button name might be “Email”, “Print”, “Cancel”

```
Widget * button = nullptr;
if (showDialog(button)) {
  printf(“Button Clicked = %s”, button->name());
}

bool showDialog(Widget *& pushButton) {      // received by reference
  if (! runDialog()) {
    return false;
  }

  pushButton = getSelectedButton();
  return true;
}
```

Parameter Passing

- Pointer to a Pointer
  - button name might be “Email”, “Print”, “Cancel”

```
Widget * button = nullptr;
if (showDialog(& button)) {
  printf(“Button Clicked = %s”, button->name());
}

bool showDialog(Widget ** pushButton) {      // received by value
  if (! runDialog()) {
    return false;
  }

  *pushButton = getSelectedButton();
  return true;
}
```

Parameter Passing

- Summary
  - pass by value should be thought of as receive by value
    - value category for the argument which is being passed can be an lvalue or an rvalue
      - lvalues will be copied
      - rvalues will be moved
  - pass by reference should be thought of as receive by reference
    - value category for the argument which is being passed depends on which type of reference is received
      - only an lvalue can be passed to an lvalue reference
      - any value category can be passed to a const reference
      - only an rvalue can be passed to an rvalue reference

Presentations

- ❑ Why CopperSpice, Why DoxyPress
- ❑ Compile Time Counter
- ❑ Modern C++ Data Types
- ❑ CsString library (unicode)
- ❑ Multithreading in C++
- ❑ Multithreading using libGuarded
- ❑ Signals and Slots
- ❑ Templates in the Real World
- ❑ What's in a Container
- ❑ Modern C++ Threads
- ❑ C++ Undefined Behavior
- ❑ Regular Expressions
- ❑ Type Traits
- ❑ C++ Tapas (typedef, forward declarations)
- ❑ C++ Tapas (typename, virtual, pure virtual)
- ❑ Overload Resolution
- ❑ Futures & Promises
- ❑ Thread Safety
- ❑ Constexpr Static Const
- ❑ When Your Codebase is Old Enough to Vote
- ❑ Sequencing, Linkage, Inheritance
- ❑ Evolution of Graphics Technology
- ❑ GPU, Pipeline, and the Vector Graphics API
- ❑ Declarations and Type Conversions
- ❑ C++ ISO Standard
- ❑ Inline Namespaces
- ❑ Lambdas in Action
- ❑ Any Optional, Variant
- ❑ CsPaint Library
- ❑ Moving to C++17
- ❑ What is the C++ Standard Library
- ❑ Attributes, Copy Elision, Time Complexity
- ❑ Qualifiers
- ❑ C++ Memory Model
- ❑ Atomics, Mutexes
- ❑ Mutexes to CsLibGuarded

Please subscribe to our YouTube Channel
https://www.youtube.com/copperspice

Applications

- KitchenSink
  - contains 30 demos and links with almost every CopperSpice library
- Diamond
  - programmers editor which uses the CopperSpice libraries
- DoxyPress & DoxyPressApp
  - application for generating source code and API documentation

Where to find CopperSpice

- www.copperspice.com
- twitter: @copperspice_cpp
- ansel@copperspice.com
- barbara@copperspice.com
- source, binaries, documentation files
  - download.copperspice.com
- source code repository
  - github.com/copperspice
- discussion
  - forum.copperspice.com