# Modern C++, From the Beginning to the Middle

Ansel Sermersheim & Barbara Geller
ACCU / C++
November 2017

# Introduction

- Where is the Beginning
- Data Types
- References
- Const Const Const
- Semantics
- Templates
- Full Example

# Where is the Beginning

- Where are you starting from?
  - if your background is from C
    - your definition of a reference may be inaccurate
    - you might think pointer and reference mean the same thing or they are the inverse of each other
  - if you started with C++98
    - your definition of a reference may be incomplete
    - you might think references are implemented as pointers
  - if your background was not C++
    - do you consider how data is passed
    - do you think about when resources are released

# Where is the Beginning

- If you think C++11 was just C++98 with a bit more stuff...
  - it should be considered a new language
  - defined new data types
  - added semantics, new value categories
  - constexpr, lambdas, smart pointers
  - added a memory model and threading library
  - sparked new interest in compiled languages

- C++ standard
  - C++98 standard is 832 pages
  - C++11 standard is 1222 pages
  - C++14 standard is 1261 pages
  - C++17 standard is 1485 pages

# Where is the Beginning

- We need to start with data types
  - can you define what a data type is?
  - what are the data types in C++?
  - what is a reference and is it a data type?
  - is a reference an idea, hype, or really important to know?
  - what are semantics?
  - is a reference the same as an lvalue reference?
  - is a forwarding reference the same thing as perfect forwarding?

# Where is the Beginning

- If you are unable to work through the following, you may not know the fundamentals of C++

  *A partially specialized templated class with an enable_if for SFINAE, containing a variadic templated method which takes a parameter pack, with a trailing return type which is deduced based on an expression decltype, then using perfect forwarding to call a policy method.*

# Data Types 101

- Definition of a data type

A data type is a classification identifying the possible values for that type and the operations which can be done on values of that type.

# Data Types 101

- Primitive or Simple Data Types
  - data types provided by the programming language
  - only one value can be associated with a variable of a primitive data type
  - very few languages allow the behavior or capabilities of primitive data types to be modified

- Examples: char, int, bool, double, float
  - the void type has an empty set of values, it is mainly used as a return type for functions

# Data Types 101

- Built In Data Types
  - programming language provides built in support

- Examples: lists, hash tables, complex numbers
  - `std::complex<double> z = 1.0 + 2i;`

# Data Types 101

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

- Examples: array, structure, class

# Data Types 101

- User Defined Data Types
  - adding classes to your program is the methodology for creating new composite data types in C++
  - another way to create a user defined data type is by declaring an enumeration type

- Examples:
  - `enum class Spices { mint, basil, salt, pepper };`

# Data Types 101

- Abstract Data Type
  - any type which does not specify an implementation
  - not necessarily an OOP concept
  - for example, a Stack has push() and pop() which have well defined behavior, however their implementation can be done in a variety of ways

  - An abstract class may not have definitions for all the methods it declares. You can not directly instantiate an abstract class. Instead, create a subclass and instantiate the child class.

# Data Types 102

- Atomic Data Types
  - no component parts which can be accessed individually
  - a type which encapsulates a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads

- Example 1:
  - a single character such as "x" is atomic

- Example 2:
  - the string "Chocolate Cake" is not atomic as it is composed of multiple individual character values

# Data Types 102

- Pointer Data Type
  - the data type of a pointer is derived from the data type or abstract data type it is pointing to
  - the data type of the pointer is a different data type from the data it points to

- `int *foo1;`
  - foo1 is a pointer to something of type int

- `Widget *var2;`
  - var2 is a pointer to something of type Widget

# Data Types 102

- Pointer Data Type
  - a pointer refers directly to another value stored elsewhere in computer memory
  - an abstract way of thinking about pointers is like a scavenger hunt
  - you proceed to the first address where you pick up the address of the real treasure is located.
  - the address of the first clue is at 1020 Palm Drive, when you arrive there is information saying the treasure is located at 1619 Pine Street

# Data Types 102

- Pointer visibility

```
class Ginger {
  ...

  private:
    std::string *m_string;
};
```

- Quiz 1
  - is the pointer private?
  - is the string m_string points to private?

# Data Types 102

- Example:
  - given an object which is a “House”
  - the address of the house is 1600
  - “1600” is stored at memory location 100

```
House *mansion;
```

- What does the receiver want?
  - if the receiver wants the data by reference or by value, then you need to pass the object

  - passing mansion passes a pointer ( passing the address 1600 )
  - passing *mansion passes the object ( passing the entire house )

# Data Types 102

- Reference Data Type
    - in C, function arguments are always passed by value
    - passing an object **by value** can be costly since it requires making a copy of the original data and then passing the copy of the data to the function or method
    - to fake pass by reference in C a pointer data type is passed to the function
    - passing **by reference** means only a reference to the data is passed and not the actual data

# Data Types 102

- Reference Data Type
  - using a pointer to implement "pass by reference" in C++ works, however it is extremely important to understand this is not a C++ reference
  - if you use a pointer to "pass by reference" you are actually passing the pointer by value
  - the called function must dereference the passed pointer to access the actual data
  - changes to the passed pointer will not affect the pointer value in the caller, but changes to the data the pointer points to, will change the original data

# Data Types 102

- Example 1:

```
House *mansion;
thing1(mansion);

void thing1(House *x) {
  print(x);
  print(*x);
}
```

- Example 2:

```
House *mansion;
thing2(*mansion);

void thing2(House &x) {
  print(x);
  print(&x);
}
```

# Data Types 102

- Example 1:

```
House *mansion;
thing1(mansion);

void thing1(House *x) {
  print(x);                  // 1600
  print(*x);                 // the house
}
```

- Example 2:

```
House *mansion;
thing2(*mansion);

void thing2(House &x) {
  print(x);                  // the house
  print(&x);                 // 1600
}
```

# Data Types 102

- Reference Data Type
  - the reference data type was added in C++98
  - references were initially introduced to just support operator overloading

  - to support pass by reference efficiently, new reference data types were added to C++11

# Data Types 102

- Reference Data Type
  - the & character can represent any of the following:
    - used in reference data types
    - address of operator
    - bitwise AND operator

# Data Types 102

- Pointers vs References
    - using a reference to an object is the same as using the original object
    - the "address of operator" will return a pointer referring to the original object
    - the C++ Standard does not force compilers to implement references using pointers

```
Widget button;
Widget & pb = button;
```

# Data Types 201

- Expressions
  - a sequence of operators and their operands which specify a computation
  - an operator with its operands, a literal, or a variable name
  - characterized by a (1) data type and a (2) value category
  - expression evaluation may produce a result (x = 2 + 3) or may generate side-effects (printf)

# Data Types 201

- Value categories
  - lvalue
  - rvalue

  - every expression is either an lvalue or an rvalue
  - an lvalue is not an rvalue and an rvalue is not an lvalue

  - the sub-categories will be explained

# Data Types 201

- Value categories are a property of an expression
    - does it have an identity
        - does the expression have a name
        - does the expression have a memory location
        - can you take the address of the expression
    - can it be moved from

# Data Types 201

- lvalue
  - typically an entity which has a name
  - the lifetime persists beyond the current expression
  - must be able to take the address using the & operator
  - has identity and can not be moved from

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

  - button is an lvalue of a pointer type
  - *button is an lvalue referring to the object button is pointing to

# Data Types 201

- Quiz 2 : values

```
int foo1 = 7;
foo1 = 9;                              // is foo1 an lvalue?


const int foo2 = 7;
foo2 = 9;                              // is foo2 an lvalue?
```

# Data Types 201

- rvalue
    - a temporary value which does not persist beyond the expression which uses it
    - may or may not have an identity
    - can be moved from

    - a literal such as 42, true, or nullptr

# Data Types 201

- Examples: values

```
int someVarA = 35;
```

  - data type of someVarA is int, it is an lvalue
  - data type of 35 is int, it is an rvalue

```
int 35 = someVarB;
```

  - this is not legal code since 35 is an rvalue and located on the left side of the expression

# Data Types 201

- References
  - lvalue reference
  - const reference
  - rvalue reference

- To understand references we ask, what does it mean to pass by value or pass by reference?

# Data Types 201

- Pass by Value
    - lvalue and rvalue, pass by value

```
class Widget{ };                    // define a class
void func(Widget pb);               // receives by value

Widget x;                           // lvalue
func(x);                            // lvalue ok

func( Widget{} );                   // rvalue ok
```

# Data Types 201

- Pass by Reference
  - lvalue reference, called func() can modify

```
class Widget{ };                    // define a class
void func(Widget & pb);             // receives by lvalue reference

Widget x;                           // lvalue
func(x);                            // lvalue ok

func( Widget{} );                   // rvalue error
```

- Pass by Reference
  - const reference, called func() can not modify

```
class Widget{ };                    // define a class
void func(const Widget & pb);       // receives by const reference

Widget x;                           // lvalue
func(x);                            // lvalue ok

func( Widget{} );                   // rvalue ok
```

# Data Types 201

- Pass by Reference
  - rvalue reference, called func() can modify however the caller can not observe the changes

```
struct Widget{ };                   // define a structure
void func(Widget && pb);            // receives by rvalue reference

Widget x;                           // lvalue
func(x);                            // lvalue error

func( Widget{} );                   // rvalue ok
```

# Data Types 201

- lvalue reference
  - caller will observe the modifications made in the called function
- const reference
  - called function can not modify the object
- rvalue reference
  - called function can modify the object
  - caller promises not to observe the changes

# Data Types 201

- rvalue reference
  - declared using &&
  - in a declaration && usually means an rvalue reference, however sometimes it means either ‘rvalue reference’ or ‘lvalue reference’
  - can be on the left side of an expression

  - C++11 extended the notion of rvalues by letting you bind an “rvalue reference” to an “rvalue”, this prolongs the lifetime of the rvalue as if it were an lvalue

# Data Types 201

- Examples: rvalue reference

```
int && func() {
  return 42;                      // returns an rvalue
}


int main() {
  int && foo = func();            // what is the “value category” of foo?
                                  // what is the data type of foo?

  int bar = foo + 3;              // is this valid? if so, what is bar?
  foo = 47;                       // does this compile?
}
```

# Data Types 201

- Example: references

```
int & func() {
  return 42;                    // 42 is an rvalue, this does not compile
}
```

   - the return type here is specifying an lvalue reference
   - however, the return expression is an rvalue, this is a compile error to ensure you do not accidentally do this

# Data Types 201

- rvalue reference
  - if you think “rvalue reference” whenever you see && in a declaration, you will misread C++
  - && might actually mean &

  - if a variable or parameter is declared to have type T && for some deduced type T, that variable or parameter is a “forwarding reference”

- Example: references

```
Widget && varA = Widget{};
auto && varB   = varA;               // && does not mean rvalue reference
```

  - varA is an lvalue (value category) of type (data type) rvalue reference to Widget

  - varB is called a “forwarding reference” which is being initialized with an lvalue
  - this means varB is deduced to be an lvalue reference
  - varB acts as if it were declared using:

```
Widget & varB = varA;
```

# Data Types 201

- Example: references

```
const Widget *foo;
someMethod(X);

void someMethod(const Widget &);
```

  - what value category does someMethod want?
  - what data type does someMethod want?
  - foo is a pointer, is it an lvalue or an rvalue?
  - what should be passed for X? (`foo`, `&foo`, `*foo`)

# Data Types 201

- Example: references

```
const Widget *foo;
someMethod(X);

void someMethod(const Widget &);
```

- what value category does someMethod want? either
- what data type does someMethod want? Widget
- foo is a pointer, is it an lvalue or an rvalue? lvalue
- what should be passed for X? *foo

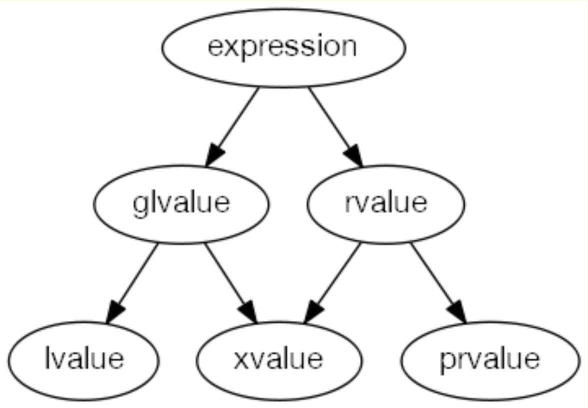
# In a Nutshell ( Definitions )

- Data Type
  - values
    - 12, true, “cake”
  - operations
    - what can be done with the data (compare, assignment, some manipulation)

- Expression
  - value category
    - lvalue, rvalue
  - data type
    - int, pointer, string, hash, lvalue reference

# Data Types - Value Categories

- C++11 additional new value categories
  - every value is either a glvalue or a prvalue, but not both
  - xvalue, an “eXpiring” value



# Data Types - Value Categories

- Rules for value categories of an expression
    - prior to C++11 the rules for distinguishing between glvalue/prvalue, the standard referred to lvalue/rvalue
    - these rules were either unintentionally wrong or contained lots of explaining and exceptions
    - the committee decided to clarify the standard and add names and definitions for glvalues and prvalues

# Data Types - Const

- Const qualifier
  - const variable
    - const int var
  - const reference
    - const Widget &var
  - const pointer
    - char *const var
  - pointer to const
    - const char *var
  - const methods
    - void someMethod() const

# Data Types - Const

- constexpr vs const
  - const means “promise not to change”
    - who promises not to change what
  - constexpr means “known at compile time”

# Data Types - Cast

- static_cast
    - always defined behavior, known at compile time
- dynamic_cast
    - always defined behavior, might fail at runtime
- const_cast
    - only used to remove const
- reinterpret_cast
    - should be called shut_up_compiler_cast
- (int)
    - should be called dangerous_cast

# Data Types - Semantics

- Semantics
  - relates to the meaning of something
  - “the lawnmower is brave”
    - the grammar or syntax is correct
    - the semantics are meaningless
  - if you misspell a command, it is a syntax error
  - when you type a legal command which does not make any sense, this is a semantic error

  - we should think about semantics when naming classes, structures, methods, functions, variables, enums, etc
  - semantics as related to the behavior of a data type
    - what does it mean when you make a copy
    - what does it mean when you assign

# Data Types - Semantics

- Different kinds of Semantics in C++
  - value semantics
  - move semantics
  - reference semantics (pointer semantics)

# Data Types - Semantics

- value semantics
  - only the value matters, not the identity or address of the object
  - usually uses the assignment operator to set a new value
    - `int x = 7;`
    - `++x;`

  - implies immutability of the object
    - an immutable object is one whose state can not be modified after it is created
      - the value is immutable, it is 7
      - the identity x, may have a changing value over time

# Data Types - Semantics

- move semantics
  - based on rvalue references
  - an rvalue is a temporary object which is going to be destroyed at the end of an expression
  - In older C++, rvalues only bind to const references
  - C++11 allows non-const rvalue references, which are references to an rvalue objects
  - since an rvalue is going to die at the end of an expression, you can steal its data
  - instead of copying it into another object, you move its data into another object

# Data Types - Semantics

- reference semantics (pointer semantics)
  - variables refer to a common value when assigned to each other or passed as parameters
  - flexibility, dynamic binding
  - objects can be large and bulky, copying them every time they are passed as parameters is slow
  - if two variables refer to the same object, modifying one of them will also make a change in the other

# Data Types - Semantics

- Examples:

```
Widget x;
Widget *y;

foo(std::move(x));        // what got moved? what semantics is this?
foo(std::move(y));        // what got moved? what semantics is this?
```

# Data Types - Pointers

- Smart Pointers, brief overview
  - abstract data type which simulates a pointer
  - provides automatic memory management
  - added in C++11
    - unique_ptr
    - shared_ptr
    - weak_ptr

  - auto_ptr
    - deprecated in C++11
    - switch auto_ptr to unique_ptr

# Templates

- Templates, defined
    - the purpose of a template is to design an entity without knowing the precise data type
    - used only at compile time to generate a class, method, function, or variable based on one or more data types
    - most of the cost for using templates is paid at compile time

# Templates

- When is a template used
  - a template is instantiated at compile time
    - for a templated class, the compiler creates a cookie cutter
    - data types in the template list are used to decide which specific instances will be required
  - at run time classes are instantiated
    - cookies are the objects or the instances of a class
    - at runtime the cookies are created and destroyed
    - only objects of the instantiated classes can be constructed

# Templates

- **Examples:** templated class with a specialized method

```
template <class T>                          // “class” or “typename”
class Widget
{
  public:
    void setName();
};

template <>                                 // required, templated class
void Widget<int>::setName()                 // specialization of a member
{
  . . .
}
```

# Templates

- **Examples:** templated class with a templated method

```
template <class T>
class Widget
{
  public:
    template<class M>
    void setName(M data);
};

template <class T>
template <class M>
void  Widget<T>::setName(M data)
{
  . . .
}
```

# Templates

- **Examples:** templated class with a class partial specialization

```
template <class T>
class Widget
{
   . . .
};

template <class X>
class Widget<std::vector<X>>
{
   . . .
};

Widget<int> foo1;                               // T is int
Widget<std::vector<int>> foo2;                  // X is int
```

# Forwarding

- Perfect Forwarding
    - a template function or method which forwards arguments while preserving the const qualifier and lvalue / rvalue category
    - rvalue reference rules are used to deduce reference types in the template instantiation
    - the called function or method will receive exactly the same arguments, with the same value categories as were passed into the function which is forwarding
    - use std::forward()

# Deduction

- Data type deduction in templates

```
template<typename T>
void func(T & someVar);

const int x = 42;
func(x);
```

  - T is deduced to be const int
  - the type of someVar is deduced as const int &

  - func() appears to take an lvalue reference but in fact it can take an “lvalue reference” or a “const reference”
    - const can be added to the T

# Deduction

- Example: rvalue reference revisited

```
template<typename T>
void func(std::vector<T> && var3);
```

  - T will be deduced to some data type
  - std::vector<T> is not a deduced data type but rather a dependent data type based on the data type of T
  - the type of var3 can only be an “rvalue reference”

# Variadic Templates

- Example:

```
template<typename ...Ts>                    // parameter pack Ts
void makeWidget( Ts ...Vs )                 // parameter pack Vs
{
  someFunc( Vs...);                         // expansion
}
```

  - the ellipsis (...) operator has two roles
    - to the left of a parameter name, it declares a parameter pack
    - to the right of an expression the ellipsis operator unpacks the parameters into separate arguments

# SFINAE

- Definition
  - substitution failure is not an error
  - occurs during template instantiation (compile time)

  - for a given T, if the compiler is unable to evaluate a template parameter then this template specialization is ignored
  - if another template that matches can be instantiated successfully no compile time error is generated

# SFINAE

- Example:

```
template <typename T, typename = void>
class Widget

template <typename T>
class Widget< std::vector<T>,
              typename std::enable_if< std::is_enum<T>::value >::type >
```

  - `is_enum<T>::value        // takes a data type,  returns a bool value`
  - `enable_if<bool>::type    // takes a bool value, returns void or compile error`

```
Widget<int>;
Widget<std::vector<Spices>>;
Widget<std::vector<int>>;
```

# Full Example

```
template<typename T, typename = void>
class Bento
{
  . . .
};


template<typename T>
class Bento<T, typename std::enable_if<std::is_move_assignable<T>::value>::type>
{
  template<typename ...ArgTypes>
  auto myMethod ( ArgTypes ...&& Vs ) ->
       decltype( T::someMethod( std::forward<ArgTypes>(Vs)... ) )
  {
    return T::someMethod( std::forward<ArgTypes>(Vs)... );
  }
};
```

# Libraries

- CopperSpice
  - libraries for developing GUI applications
- CsSignal Library
  - standalone thread aware signal / slot library
- CsString Library
  - standalone unicode aware string library
- libGuarded
  - standalone multithreading library for shared data

# Please subscribe to our Youtube Channel

- Presentations
  - Why DoxyPress
  - Why CopperSpice
  - Compile Time Counter
  - Modern C++
  - CsString
  - Multithreading in C++
  - Next video available on Nov 16

https://www.youtube.com/copperspice

# Where to find CopperSpice

- www.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- source, binaries, documentation files
  - download.copperspice.com

- source code repository
  - github.com/copperspice

- discussion
  - forum.copperspice.com