C++ 11 Key Features

TOPICS

This module reviews the some of the more useful or impactful features of C++ 11.

- default
- delete
- alias
- unions
- initialization lists
- constant expressions

DEFAULT

- Designates a method as a default. The best example is the default constructor. You can actually lose your default constructor by adding a constructor with parameters.
- The default constructor can be added back as shown to the right. You can also use the default keyword to change the attributes of a compiler generated default method.

```
class ClassX{
public:
 ClassX(double) {}
 ClassX() = default;
 ClassX(const ClassX &rhs) = default;
};
```

DELETE

- The *delete* keyword means a particular method cannot be invoked or callable.
- Delete can be applied to a generated method or an explicit method.

```
template<typename T>
void DoSomething(T _arg) {}
void DoSomething(int arg) = delete;


void main()
{
  DoSomething(1.5);
  // this will not compile
  DoSomething(1);
}
```

DELETE – ANOTHER EXAMPLE

```
class ClassA
{
public:
    ClassA(double) {}
    ClassA(int) = delete;
    ClassA() = delete;
};

void main()
{
    ClassA obj1(12.0);
    ClassA obj2(1);  // does not compile
    ClassA obj3;     // does not compile
}
```

.

INHERITING CONSTRUCTORS

- This feature streamlines the invocation of base class constructors in the child class.
- Inheriting constructors allows a child class to directly inherit base class constructors.
- This is particularly useful when the child has no initialization otherwise.
- With inheriting constructors, the child class can override any inherited constructor and reimplement.

```
class ClassA {
public:
  ClassA() {}
  ClassA(int) {}
};
class ClassB : public ClassA {
public:
     using ClassA::ClassA;
};
int main()
{
   ClassB b(5); // ClassA(int) C'tor
}
```

DELEGATING CONSTRUCTORS

- A constructor can delegate to a constructor of the same class through the colon initialization list.
- More efficient than creating a member function for assignment.

```
struct Rational {
   Rational(int n) : Rational(n, 1) {};
   Rational( int n, int d )
   : _num(n), _denom(d == 0 ? 1 : d) {};

   void Display( ) {}
private:
   double _num, _denom;
};
```

IMPLICIT CONSTRUCTORS AND OPERATORS

Certain C++ class methods can be called implicitly when not expected, which has been the bane of many developers. This is particularly true of constructors and operator methods.

- Conversion constructor
- Copy constructor
- Default constructor
- Assignment operator

```
class IntWrapper {
public:
      IntWrapper(int data){
        _data = data;
      }
      int GetData(){
         return _data;
      }
private:
      int _data;
};

IntWrapper Obj = 10;
```

PREVENT IMPLIED INVOCATION

Preface a member method with the *explicit* specifier to prevent implicit invocation.

```
class IntWrapper {
public:
    explicit IntWrapper(int data){
    _data = data;
}
int GetData(){
    return _data;
}
private:
    int _data;
};
Int main()
{
    Int impliedObject = 10;  // error
}
```

USING ALIAS

Better known as *type alias*.

- Replacement for typedef
- Reduce typing
- Add context to code

```
// FP is a synonym for a pointer to a function
// taking an int and a const std::string&
// and returning int
typedef int (*FP)(int, const std::string&);

// same meaning as above
using FP = int (*)(int, const std::string&);
```

UNIONS

- Prior to C++ 11 unions could not include members that are types with non-trivial members, such as explicit constructors.
- That restriction has been removed.
- If the non-static data member has a non-trivial constructor, the union must also explicitly have that constructor.

```
union Integer{

     Integer() {} // important
     RationalNumber rational;


     int numerator;
     int denomintor;
};

int main(){

     Integer number;
     number.numerator = 30;
     number.denomintor = 15;

     cout << number.rational.GetNumerator()
      << " / "
      << number.rational.GetDenominator()
      << endl;

     return 0;
}
```

DISCRIMINATING UNIONS

- Discriminating unions are a combination of a type and an anonymous union.
- The structure contains a member that indicates the active member.

```
class DiscriminatedUnion {
public:
    DiscriminatedUnion(int var)
    : memtype(_int), memberi(var) {}

    DiscriminatedUnion(double var)
    : memtype(_double), memberii(var) {}

    enum { _int, _double} memtype;
    union {
        int memberi;
        double memberii;
     };
};
```

INITIALIZATION LISTS - 1/2

- C++ 11 has uniform initialization, instead of a patchwork of solutions.
- Get ready for curly braces!

```
class Musician{
public:
     string first;
     string last;
};

int main() {
  Musician unknown{}; // default c'tor
  Musician beatle = { "Ringo", "Starr" };
  int scores[] = { 100, 85, 95, 79 };
  int *pInt = new int[4] {80, 74, 92};
}
```

INITIALIZATION LISTS – 2/2

- Initialization can be nested.
- Nested initialization is used to set the structure values.

```
struct Point{
    int x;
    int y;
};


class Locations {
private:
   Point _coordinates[3];

public:
   Locations()
   : _coordinates{ {1,2}, {3,4}, {5,6} }{}

};
```

INITIALIZER_LIST TYPE

The initializer_list type can be assigned an initializer list (curly braces).

- Include initializer_list header file.
- Several STL containers have a constructor with a initializer_list parameter.
- Can be used with user defined types
- Has a begin, end, and size methods.

```
#include <initializer_list>
#include <vector>

class ClassA {
public:
    ClassA(initializer_list<int> list)
    {
        for (auto mem : list)
        {  _elements.push_back(mem); }
    }
private:
    vector<int> _elements;
};
int main() {
    ClassA obj({ 5,9,6,4 });
}
```

MUTABLE

- The keyword **mutable** is used to allow a data member of const object to be modified.
- When we declare a function as const, the this pointer passed to function becomes const.
- Adding mutable to a variable allows a const pointer to safely change members.

```
class DeviceMonitor{

private:
    const double _threshold = 1.2;
    mutable double _charge;
    double _potential, _capacitance;

    void CheckCapacitance() const {

        // possibly to safely alter charge

    }

public:
    bool IsCharged() const {
        CheckCapacitance();
        return (_capacitance < _threshold);

    }
};
```

CONSTANT EXPRESSION

- The keyword **constexpr** was introduced in C++11 . Like const, it can be applied to identifiers: A compiler error is raised when any code attempts to modify the value.
- **constexpr** can be applied to functions and class constructors.
- A constexpr integral value can be used wherever a const integer is required, such as in template arguments and array declarations.
- A constexpr function is one whose return value is computable at compile time when consuming code requires it.

```
constexpr long exp(int x, int n){

  return n == 0 ? 1 :
              n % 2 == 0 ? exp(x * x, n / 2):
              exp(x * x, (n - 1) / 2) * x;
}

int main(){

  constexpr long z = exp(2, 3);
  long ar[z];

  ...
}
```