TPF Users Group Spring 2005

Get Ready For Standard C++!

Name : Edwin W. van de Grift
Venue : Applications Development Subcommittee

AIM Enterprise Platform Software

IBM z/Transaction Processing Facility Enterprise Edition 1.1.0

ISO C++

- C++ has been standardized by ANSI (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German national standards organization), several other national standards bodies, and ISO (International Standards Organization)

- ISO/IEC 14882:1998
  - Available through the National Committee for Information Technology Standards' electronic store.
  - http://www.ncits.org/

ISO C++?

- On TPF4.1
  - The version of the IBM C++ compiler that is supported on TPF4.1, only implements a subset of the ISO C++ standard.
- On z/TPF
  - The GNU Compiler Collection (GCC) C++ compiler -- G++ -- does support the ISO C++ standard.

So, what parts of C++ didn’t we have on TPF4.1?

- Namespaces
- New cast operations
- Booleans
- Explicit and mutable specifiers
- Real-Time Type Information
- Standard exceptions
- Universal character names
- Standard library (including STL)

Namespaces

- A namespace is an optionally named declarative region.
  - The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace.
  - The definition of a namespace can be split over several parts of one or more translation units.
  - A name declared outside all named namespaces, blocks and classes has global namespace scope.

Namespaces

- Defining a namespace:

```
namespace MyApp {
   enum Rc {Ok, Nok};
   Rc myFunction();
}
```

```
MyApp::Rc rc = MyApp::myFunction();
```

- Using a namespace:

```
using namespace MyApp;
Rc rc = myFunction();
```

Why Namespaces?

- The way to avoid name collisions (of variables, types, classes or functions).
- Especially in big application environments.
  - TPF installations are an excellent example of this!

New Cast Operations

- Const and volatile cast
- Reinterpret cast
- Static cast
- Dynamic cast

C++ Cast Operations

- C++ standard cast operations that provide finer control over casting than previous cast operations.
- The dynamic_cast<> operator provides a way to check the actual type of a pointer to a polymorphic class.
- Casts all perform a subset of the casts allowed by the classic cast notation.
  - For example, const_cast<int*>v could be written (int*)v.
- Casts categorize the variety of operations available to express the programmer's intent more clearly
- Casts allow the compiler to better check the code.

Const and Volatile Cast

- Const and volatile cast can be used to change the const or volatile qualifiers of objects and pointers.
  - The volatile keyword specifies that the value associated with the name that follows can be modified by actions other than those in the user application.
  - When a name is declared as volatile, the compiler reloads the value from memory each time it is accessed by the program.

Const and Volatile Cast

```
const int i(42);

. . .

int* j = const_cast<int*>(&i);
```

Reinterpret cast

- Reinterpret cast changes the interpretation of the value of an expression.
- Is engineered for casts that yield implementation dependent results.
  - Not likely to use very often.
- It can be used to convert between
  - Pointer and integer types.
  - Unrelated pointer types.
  - Pointer-to-member types.
  - Pointer-to-function types.

Static cast

- When converting from a type with a small value space to a larger value space, an implicit cast can be employed as there is no danger of losing data.
- On the other hand, when converting from a larger type to a smaller type, it is possible for the data to be truncated.
  - Therefore compilers require confirmation in the form of a cast.
- This is referred to as a static cast, since only static type information is used in determining the conversion behavior.
- A static cast is often considered dangerous.
  - Compiler places the responsibility of ensuring the safety of the cast squarely in the hands of the programmer.
- Closest in meaning to conventional C-style casts.

Static Cast

```
int i;
. . .
short s = static_cast<short>(i);
```

Dynamic cast

- The dynamic type cast will convert a pointer or reference to one class into a pointer or reference to another class.
  - That second class must be the fully-derived class of the object, or a base class of the fully-derived class.
- Under virtual inheritance and multiple inheritance of a single base class, the dynamic cast must be able to identify a unique match.
  - If the match is not unique, the cast fails.
    - For pointer types, if the specified class is not a base of the fully derived class, the cast returns 0.
    - For reference types, if the specified class is not a base of the fully derived class, the cast throws a bad_cast exception.

Dynamic Cast

```
class B { ... };
class C : public B { ... };
class D : public C { ... };

void f(D* d) {
   C* c = dynamic_cast<C*>(d); // Ok: C is a direct base class
                               // c points to C subobject of d
   B* b = dynamic_cast<B*>(d); // Ok: B is an indirect base class
                               // b points to B subobject of d
}
```

Boolean

- The bool keyword is a built-in type.
- A variable of this type can have values true and false.
  - Conditional expressions have the type bool and so have values of type bool.
  - For example, i != 0 now has true or false depending on the value of i.
- The values true and false have the following relationship:
  - !false == true
  - !true == false

Explicit Specifier

- Constructors declared explicit will not be considered for implicit conversions.
- Can only be applied to in-class constructor declarations.

```
class X {
public:
  explicit X(int);
};
```

Mutable Specifier

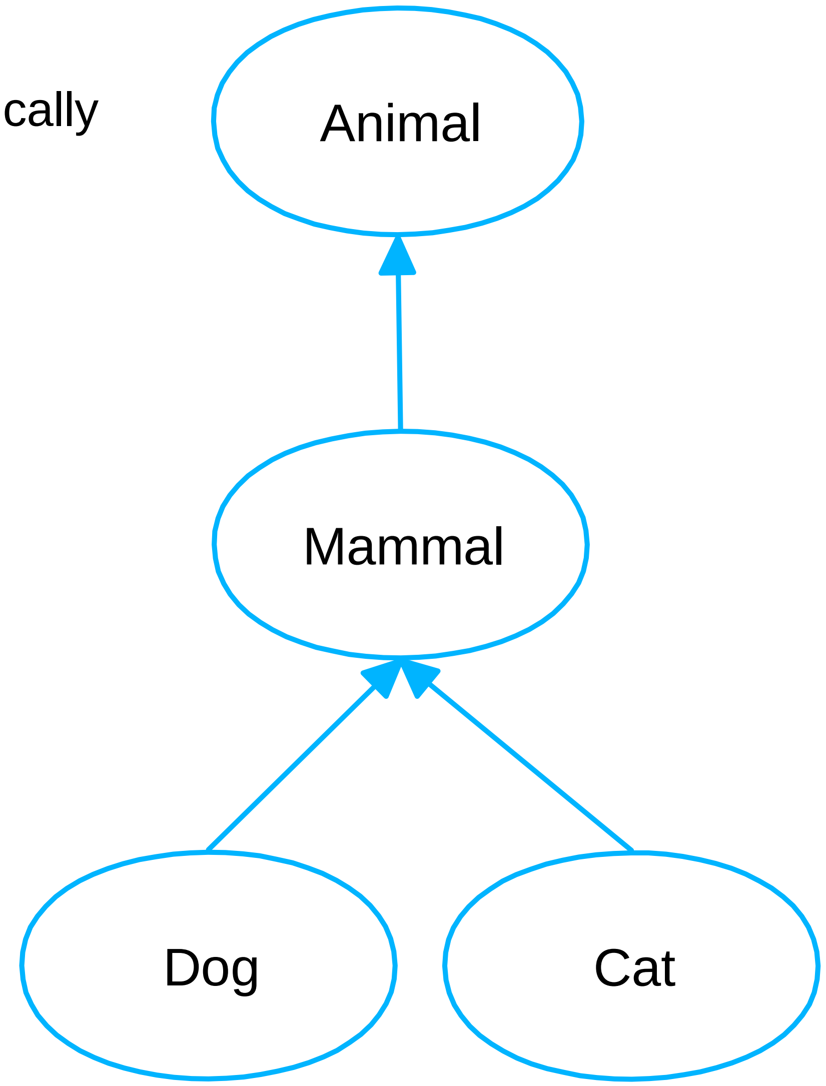
- Declare a member non-constant even if it is a member of a const object.
  - Use of mutable data members might seem like cheating because it enables a const function to modify an object's data members.
  - However, when used properly, mutable improves your code quality as it hides implementation details from users without resorting to dubious things like const_cast<>.
  - Most common usage is in object that are logically constant, and bitwise are not.

```
class String {
public:
  size_t length() const;
private:
  char* _data;
  mutable size_t _length;
  mutable bool _lengthIsValid;
};
```

```
size_t String::length() const {
  if(!_lengthIsValid) {
    _length = strlen(_data);
    _lengthIsValid = true;
  }
  return _length;
}
```

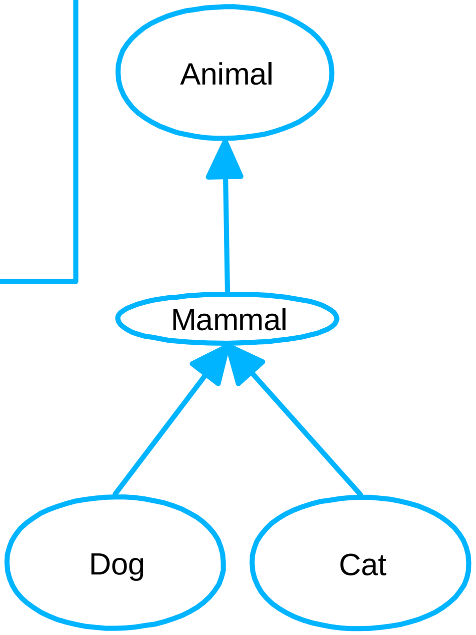
RTTI – Real-Time Type Information

- Ability to determine the type of an object dynamically
  - At runtime versus at compile time
- Three components:
  - Operator typeid
  - Operator dynamic_cast
  - Class std::type_info



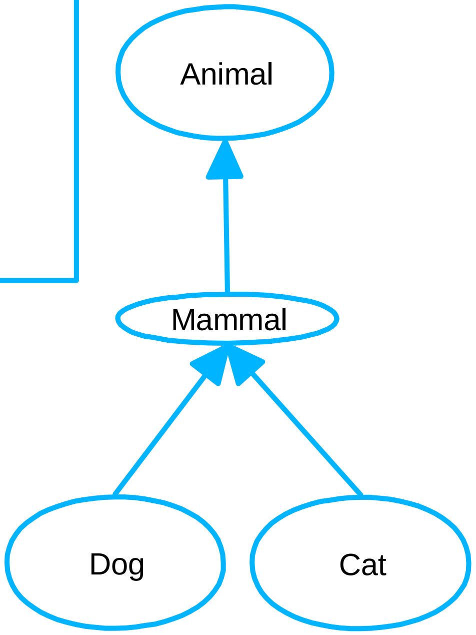
Sample Snippets: Animal

```
class Animal {
public:
  Animal(const Weight& w, const Age& a) : _w(w), _a(a) {}
  virtual ~Animal() {}
  virtual Age& getAge() const { return _a; }
  virtual void eat() = 0;
private:
  Weight _w;
  Age _a;
};
```



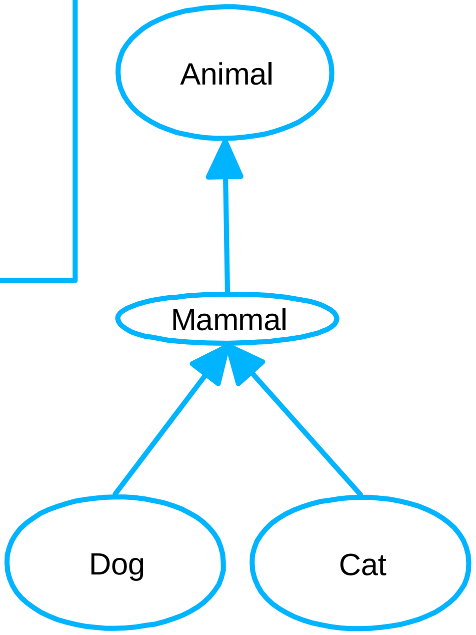
Sample Snippets: Mammal

```
class Mammal : public Animal {
public:
  Mammal(const Weight& w, const Age& a) : Animal(w,a) {}
  virtual ~Mammal() {}
  virtual void eat() {};
  virtual void speak() = 0;
private:
};
```



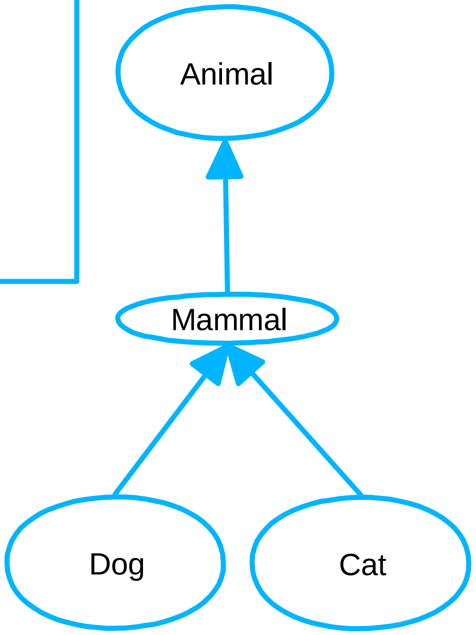
Sample Snippets: Dog

```
class Dog : public Mammal {
public:
  Dog(const Weight& w, const Age& a) : Mammal(w,a) {}
  virtual ~Dog() {}
  virtual void speak() { cout << "Ruff\n"; }
private:
};
```



Sample Snippets: Cat

```
class Cat : public Mammal {
public:
  Cat(const Weight& w, const Age& a) : Mammal(w,a) {}
  virtual ~Cat() {}
  virtual void speak() { cout << "Meow\n"; }
private:
};
```



RTTI

- RTTI is applicable solely to polymorphic objects.
  - Class must have at least one virtual-member function in order to have RTTI apply to its objects.
- For every distinct type, C++ instantiates a corresponding std::type_info object.
  - All objects of the same class share a single type_info object.
  - The most commonly used member functions are name() and operator==.

std::type_info

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();1
    bool operator==(const type_info& rhs ) const;
                                                  2
    bool operator!=(const type_info& rhs ) const;
    bool before(const type_info& rhs ) const; 3
    const char* name() const;
  private:
    type_info(const type_info& rhs );
                                               4
    type_info& operator=(const type_info& rhs);
  };
}
```

std::type_info

1. Class type_info can serve as a base class
2. Objects can be compare
3. Ordering
4. Class type_info objects cannot be copied

Operator typeid

- Operator typeid takes either an object or a type name as its argument and returns a matching type_info object.

```
void foo(Animal& a) {
  if( typeid(a) == typeid(Mammal) ) {
    cout << typeid(a).name() << " says: ";
    a.speak();
  }
}
```

```
Dog says: Ruff
Cat says: Meow
```

Dynamic cast

- A dynamic_cast<> expression to verify the type of an object can take either a pointer or a reference for an argument.

```
dynamic_cast<T*>(p)
```

Returns a T pointer or zero

```
dynamic_cast<T&>(r)
```

Returns a T reference or throws

a bad_cast exception

Dynamic cast of a pointer

- After a dynamic_cast<> expression of a pointer always test the pointer for the result.

```
MyType* myType = dynamic_cast<MyType*>(p);
if(myType) {
   // dynamic_cast succeeded
} else {
   // dynamic_cast failed
}
```

Dynamic cast of a reference

- Always place a dynamic_cast<> expression of a reference inside a try block and include a catch statement to handle std::bad_cast exceptions.

```
try {
   MyType& myType = dynamic_cast<MyType&>(p);
}
catch(std::bad_cast& e) {
   // dynamic_cast failed
}
   // dynamic_cast succeeded
}
```

Performance of typeid versus dynamic_cast<>

- From design point of view, dynamic_cast<> should be preferred to typeid because it enables more flexibility and extensibility.
- However, the runtime overhead of typeid can be less expensive than dynamic_cast<>, depending on the operands.
  - Invoking operator typeid is a constant time operation.
    - It takes the same length of time to retrieve the runtime type information of every polymorphic object, regardless of the object's derivational complexity.
  - Using dynamic_cast<> is not a constant time operation.
    - It has to traverse the derivation tree of the operand until it has located the target object in it.

Standard Exceptions

- C++ defines a hierarchy of standard exceptions that are thrown at runtime when abnormal conditions arise:
  - The standard exception classes are derived from std::exception (defined in the <stdexcept> header).
  - std::bad_alloc is defined in <new>.
  - std::bad_cast is defined in <typeinfo>.
  - Other exceptions are defined in <stdexcept>.
- This hierarchy enables the application to catch these exceptions in a single catch statement.

Universal Character Names

- The C++ Standard also allows an implementation to offer extended source character sets and extended execution character sets. Furthermore, those additional characters that qualify as letters can be used as part of the name of an identifier.
  - Thus, a German implementation might allow you to use umlauted vowels and a French implementation might allow accented vowels.
- A universal character name begins either with \u or \U.
  - \u is followed by 8 hexadecimal digits
  - \U by 16 hexadecimal digits
  - Digits represent the ISO 10646 code for the character

```
int k\u00FChlNicht(42);
```

```
int kühlNicht(42);
```

Standard Library

- It is big, it is huge, and it is the standard!
- All in the std namespace
- Almost everything is a template
- New header file names

Standard Library Header File Names

- Old headers deprecated, but still supported.
  - And will be for many years.

- C++ Headers: No file extention
  - All is in namespace std
- C Headers: Prepended with a "c" and no file extention
  - All is in namespace std

$$\begin{tikzcd}
\boxed{\text{iostream.h}} \arrow[r] & \boxed{\text{iostream}} \\
\boxed{\text{stdio.h}} \arrow[r] & \boxed{\text{cstdio}}
\end{tikzcd}$$

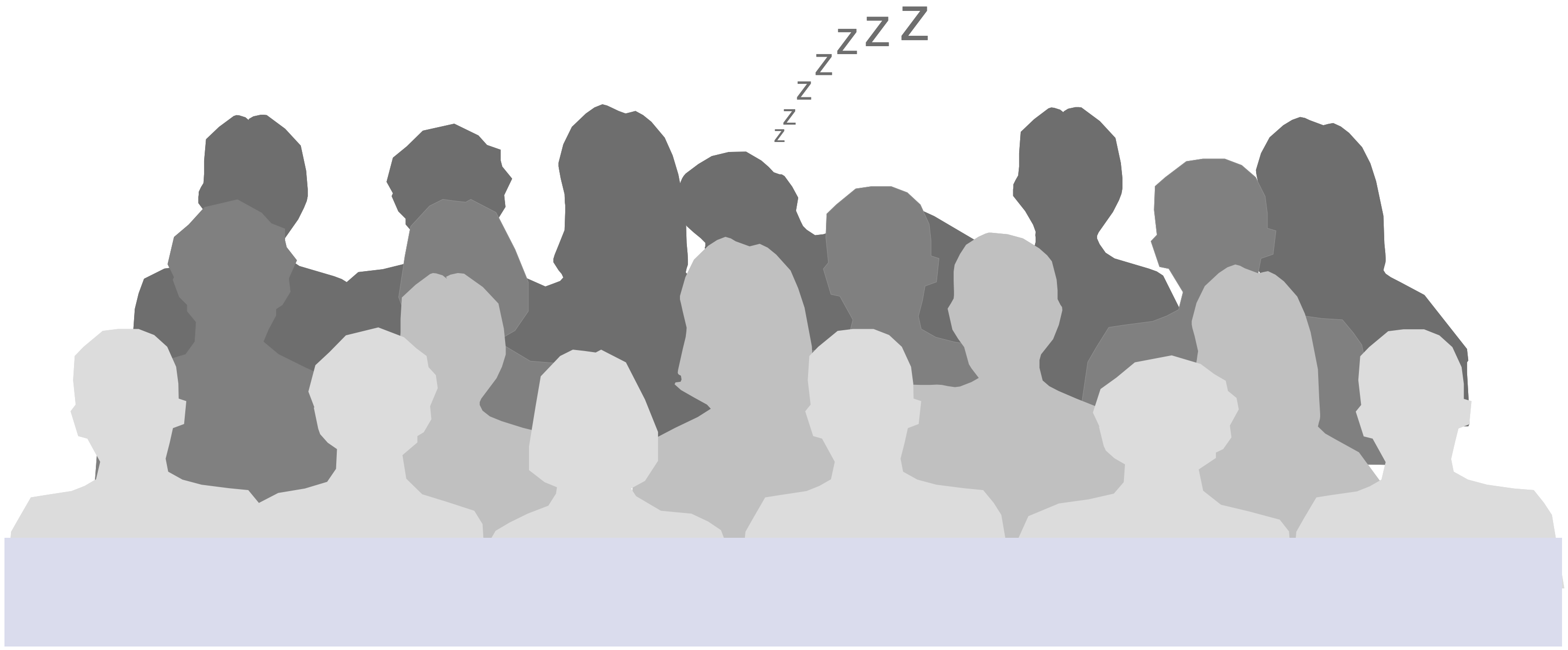
Standard Library Contents

- The standard C library
- Iostreams
- Strings
- Containers
- Algorithms
- Support for internationalization

Getting ready for standard C++

- The compilers are far better assembler programmers than humans will ever be.
- Take advantage of 64 bit architecture.
  - It does not make sense to try to outperform the compiler, or have to worry about storage mapping.
  - You shouldn’t want to program in assembler anymore.
- Take advantage of standard C++
  - Universities and colleges teach students standard C++.
  - Why “un-train” them?
- Get training!
  - Having C programmers design and write standard C++ programs is NOT a good idea.
    - This does work when using the C++ compiler to merely “optimize” C code: “A Better C”, but C++ has so much more to offer!
  - Suboptimal code is much more expensive than training.

Questions?



Trademarks