COMP 3522

Object Oriented Programming in C++

Week 8 Day 1

Agenda

1. Strict weak ordering
2. Casting
3. Enums
4. Static

COMP 3522

Note: Strict weak ordering

- Almost all C++ STL functions/containers require the ordering to satisfy the standard mathematical definition of a strict weak ordering.
- If your logic follows all three rules, it satisfies strict weak ordering
- Let **lessThan**(left, right) be a comparison function. The compare function is in strict weak ordering iff:

1. IRREFLEXIVITY: **lessThan(x, x) == false**
   - If x is passed into both parameters, the expected output is false
   - x can not be less than itself

Note: Strict weak ordering

2. ANTISYMMETRY: **if lessThan(x, y) then !lessThan(y, x)**
   - If x is less than y, then y can not be less than x

3. TRANSITIVITY: **if lessThan(x, y) and lessThan(y, z) then lessThan(x, z)**
   - If x is less than y, and y is less than z, then x is less than z

Defining strict weak ordering

- We can do this in three ways:

1. Define **operator<**(const Obj& lhs, const Obj& rhs) inside the class
2. Define a **custom comparison function** that is a binary predicate (takes two elements and returns a boolean)
3. Implement **operator( )( )** as a comparison function in a separate struct
   - Functors! We'll talk about this later in the term

CASTING

Casting

- Java and C++ are **strongly typed** languages
- The type of each object (variable or constant) is defined at **compile time** and cannot be changed during execution
- We can think of an object as:
  1. Bits in memory
  2. A **type that gives these bits meaning**.

Widening

What will be printed?

```
short small = 16;
int  medium  =  64;
medium  =  small;

cout << medium;
```

Output: 16

Narrowing

How about this? What will be printed?

```
short small = 16;
long  large  =  32768;
small =  large;
cout << small;
```

Output: -32768 //32768 too large for short, wraps to -32768

What if we KNOW the value in large is small enough?

Truncating with floats

What about floating point numbers?

```
long large = 64;
double  largeFloat  =  3.14;
large = largeFloat;
cout << large;
```

Output: 3

Casting is ugly in C

- We can cast a pointer or arithmetic type to an integer type
- We can cast an arithmetic type to a floating point type
- We can cast a pointer or arithmetic type to another pointer type

- There is lots of room for error here
- No check at runtime is performed to see if the cast is correct

So we try not to cast in C

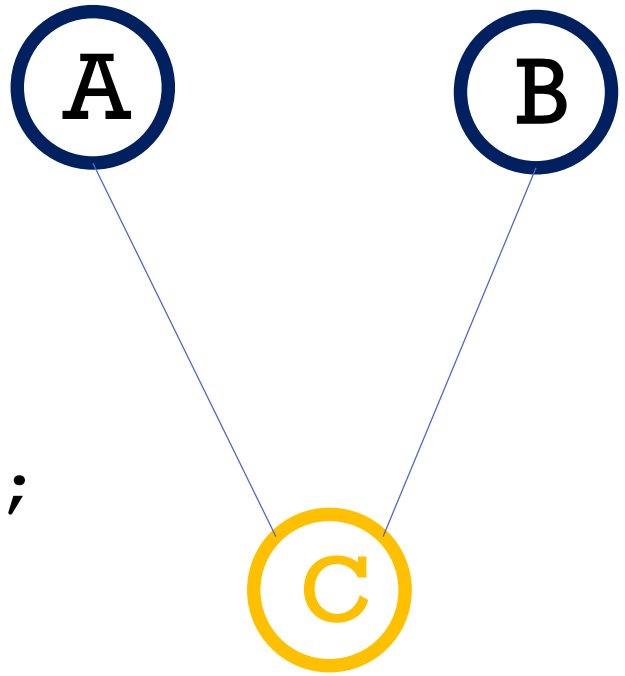
- **Notation** is hard to spot (cast to type)
- We can basically convert **anything to anything**
- There is lots of room for **error** here
- But programmers do it anyway.

Vocabulary check

- **Upcasting**: cast pointer/reference from a derived class to a base class
  - Always allowed for public inheritance (is-a relationship)
- **Downcasting**: casting pointer/reference from a base class to a derived class
  - Not allowed without explicit type cast
- **Cross-casting**: casting pointer/reference from a base to a sibling class

Cross-casting?!

Consider this inheritance hierarchy:




```
A* an_A = new C;
B* a_B = dynamic_cast<B*>(an_A);
```

We will see this madness a few times this term.

C++ casting can be safer

Bjarne Stroustrup views the C cast operator as relaxed, not strict

He introduced some new **casting operators** to C++:

1. dynamic_cast
2. const_cast
3. static_cast
4. reinterpret_cast

Casting between base and derived classes

- Let's look at **upcasting**
- Casting up from a derived to a base class is always possible when there are no ambiguities
- Can be **implicit**, i.e., a function that accepts a parameter of the base class accepts all sub-classes without the need for explicit conversion
- Upcasting is synonymous with **polymorphism**.

Upcasting example page 1 of 2

```
struct A
{
    virtual void foo(){}
    virtual ~A(){}
    int value;
};
struct B : public A
{
    float some_value;
    int foob() { return 22; }
};
```

Upcasting example page 2 of 2

```
void f(A a) { … } // NOT POLYMORPHIC (SLICING)
void g(A& a) { … }
void h(A* a) { … }

B b;
f(b); // OK, BUT BE CAREFUL! THIS IS SLICING
g(b); // OK
h(&b); // OK
```

These are all examples of implicit upcasting – the object b is converted to an object of base type A automatically.

upcast_and_slice.cpp

Casting between base and derived classes

- Let's move to **downcasting**
- This is the conversion of a pointer/reference to a sub-type pointer/reference
- When the actual referred-to object is not of that sub-type the behaviour is **undefined**

Casting between base and derived classes

If we need to downcast, we should ask ourselves why:

1. How do we make sure the object is really that sub-type?
2. What should we do if the object cannot be downcast?
3. Why not just overload functions for the two types?
4. Can we redesign our classes so that we can accomplish our task with late binding and virtual functions?

If we still need a downcast, there are two good choices in C++.

Dynamic and static casting

- C++ offers a dynamic cast and a static cast between classes in an inheritance hierarchy

dynamic_cast<target_type pointer or reference>(variable)

static_cast<target_type pointer or reference>(variable)

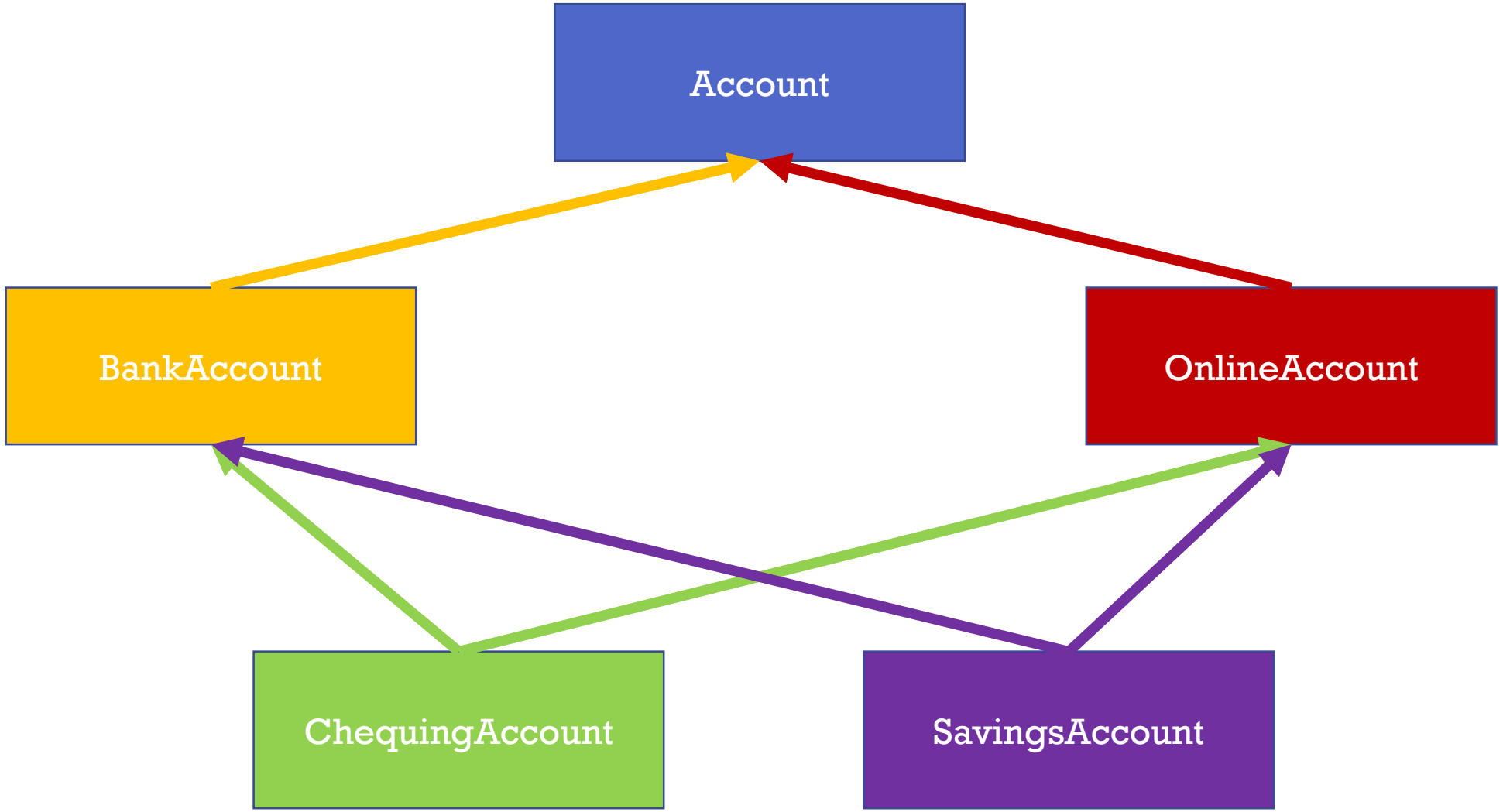
Dynamic cast

- Casts a pointer of one type to a pointer of another type within an inheritance hierarchy.
- Performs a **run-time test** to determine whether the actually casted object has the target type or a sub-type thereof
- Allowed with **pointers and references to polymorphic types** (recall polymorphic types are types with at least one virtual member function)
- **Returns nullptr on failure**
  - Note: cast to a reference target type instead of a pointer target type to throw a bad_cast exception

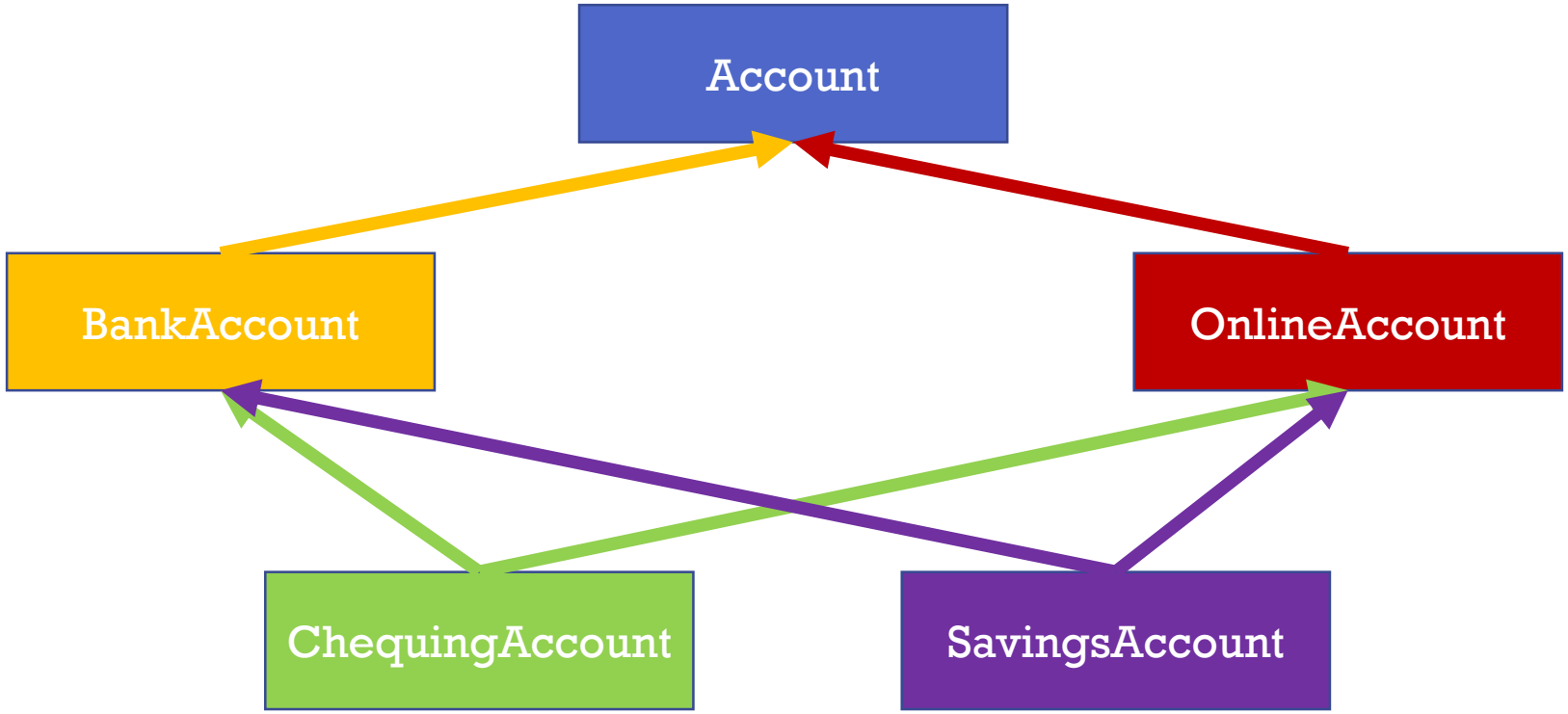
Dynamic cast example

- Suppose we have an Account class
- Account is derived by BankAccount and OnlineAccount
- ChequingAccount has 2 base classes, BankAccount and OnlineAccount
- SavingsAccount has 2 base classes, BankAccount and OnlineAccount

Dynamic cast example

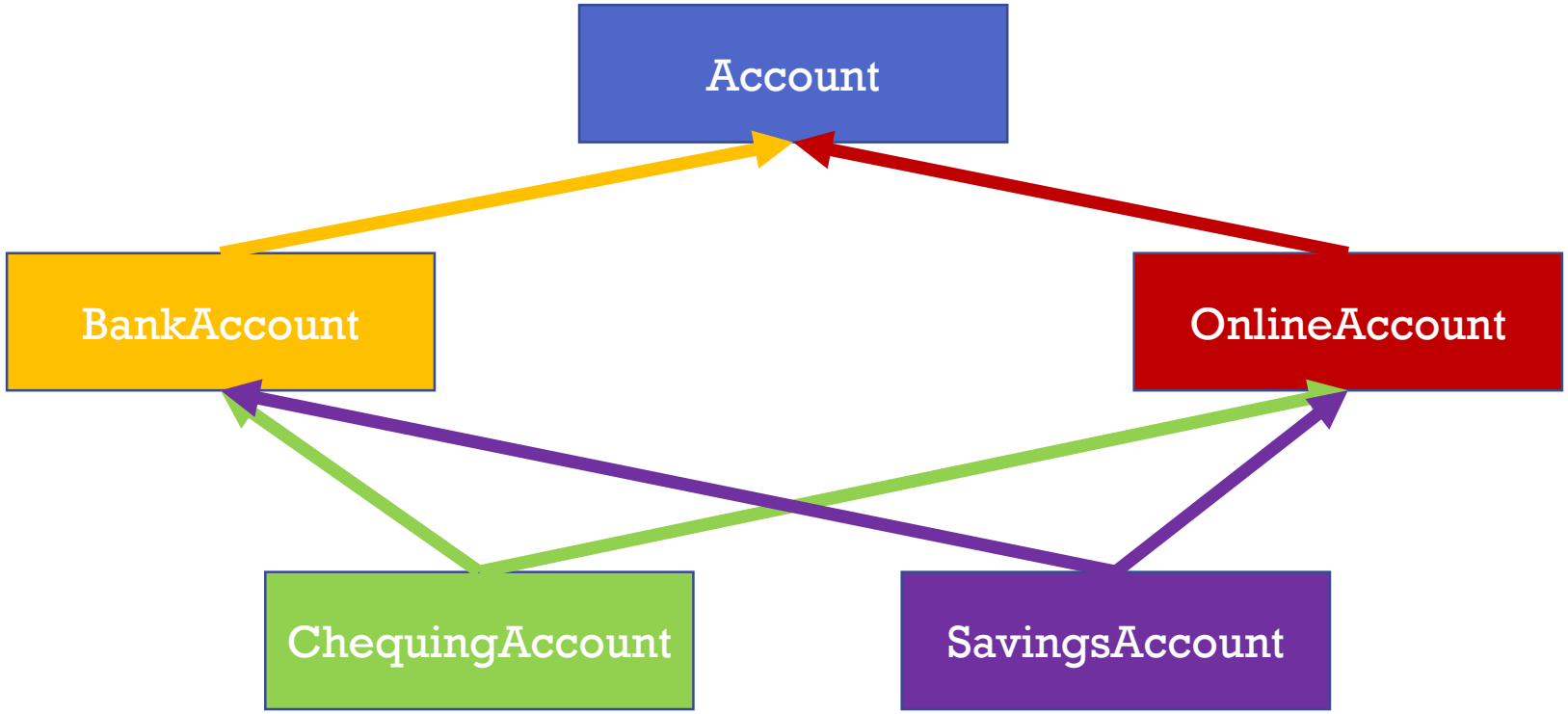


Upcast



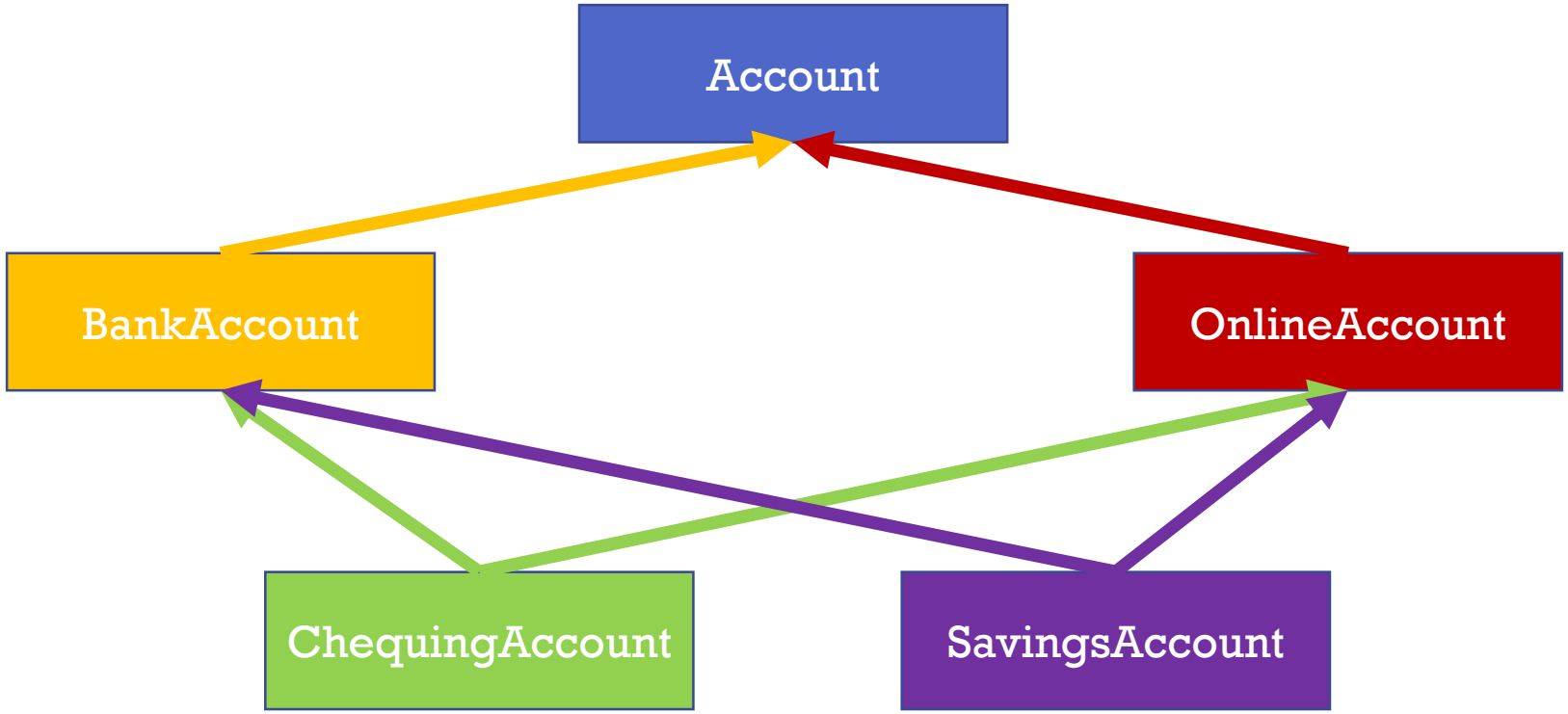
- ChequingAccount c;
- Account * account_pointer = dynamic_cast<Account *>(&c);
  // Pointer upcast

Upcast



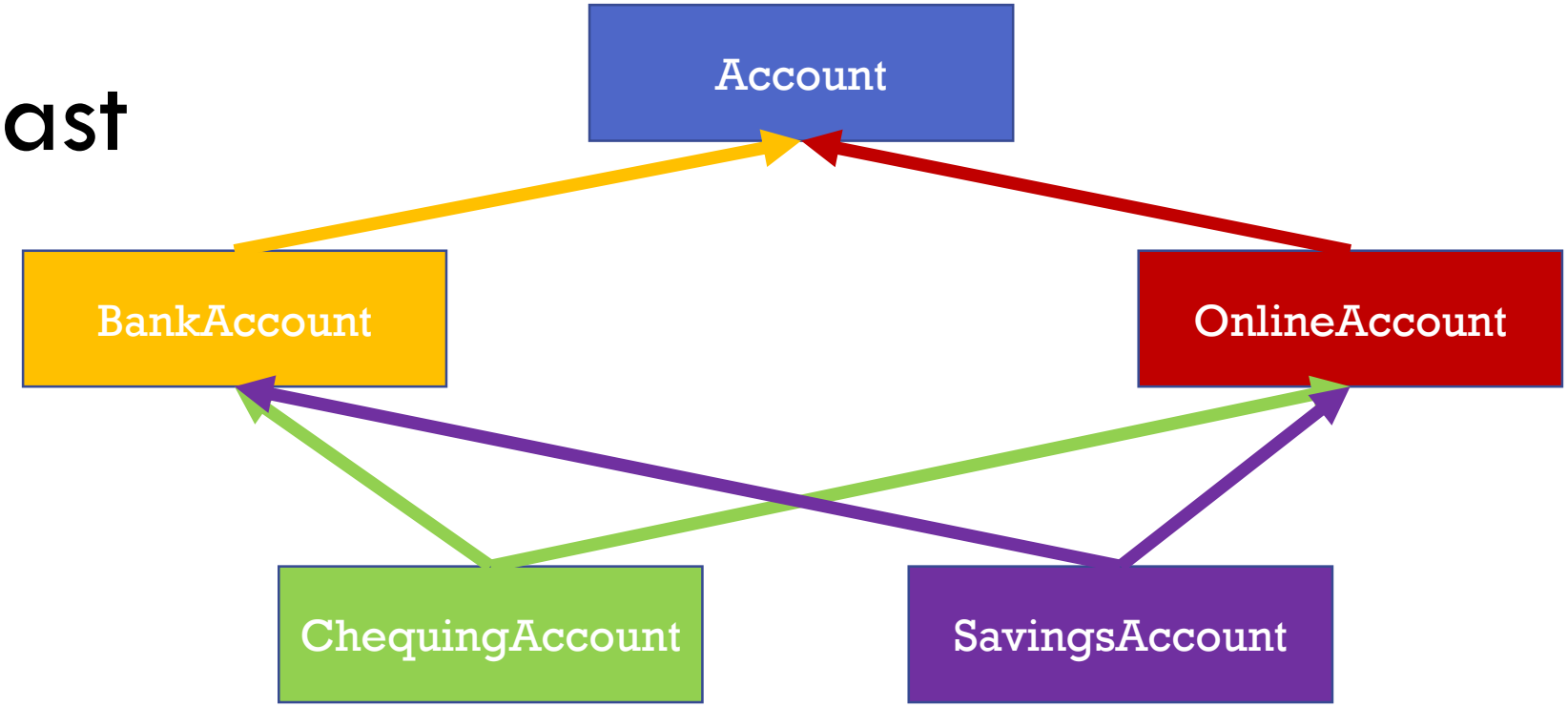
- BankAccount b;
- account_pointer = dynamic_cast<Account *>(&b); // Pointer upcast

Upcast

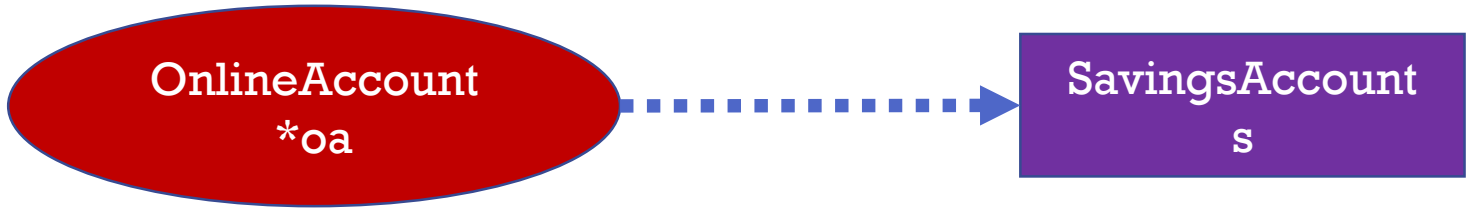


- OnlineAccount o;
- account_pointer = dynamic_cast<Account *>(&o); // Pointer upcast

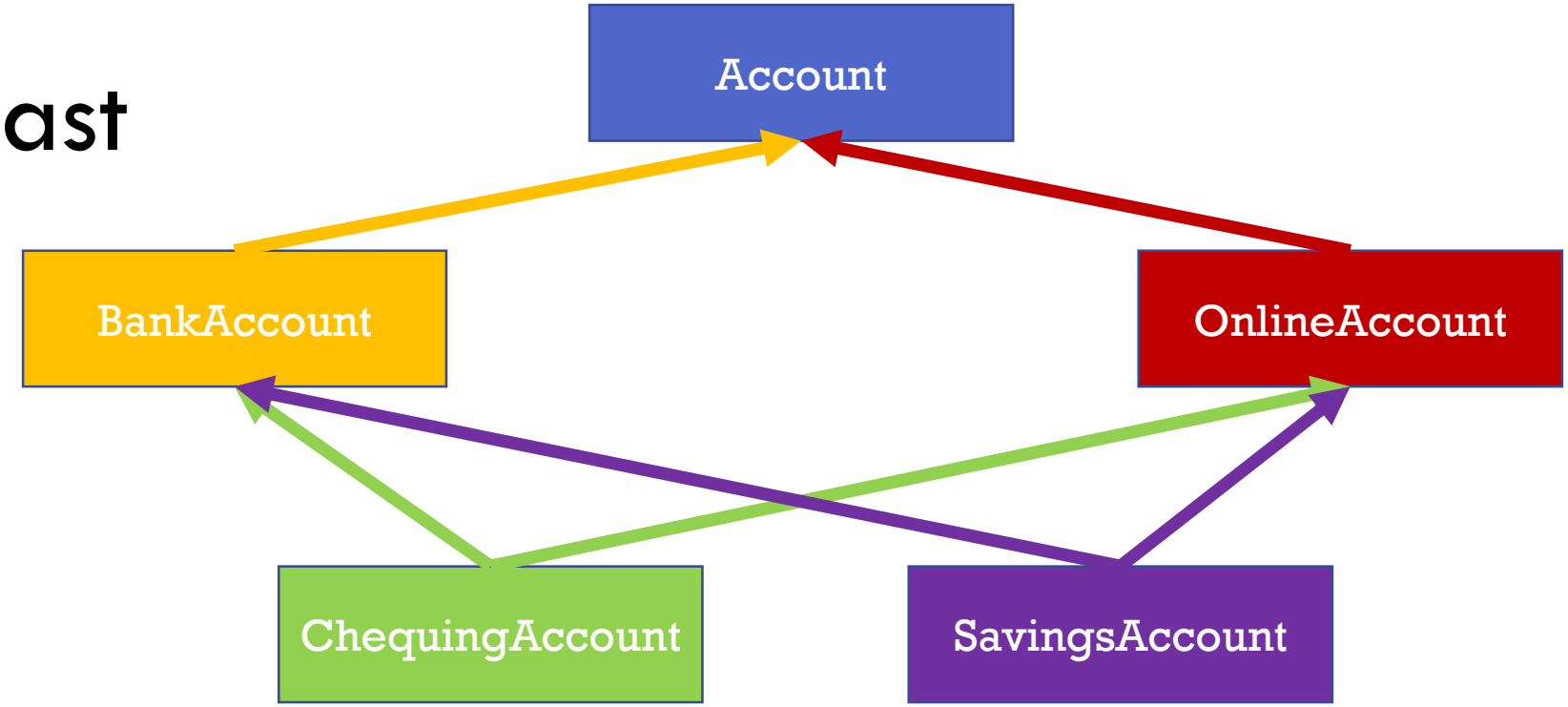
Downcast



- SavingsAccount s;
- OnlineAccount * oa = dynamic_cast<OnlineAccount *>(&s); // Pointer upcast

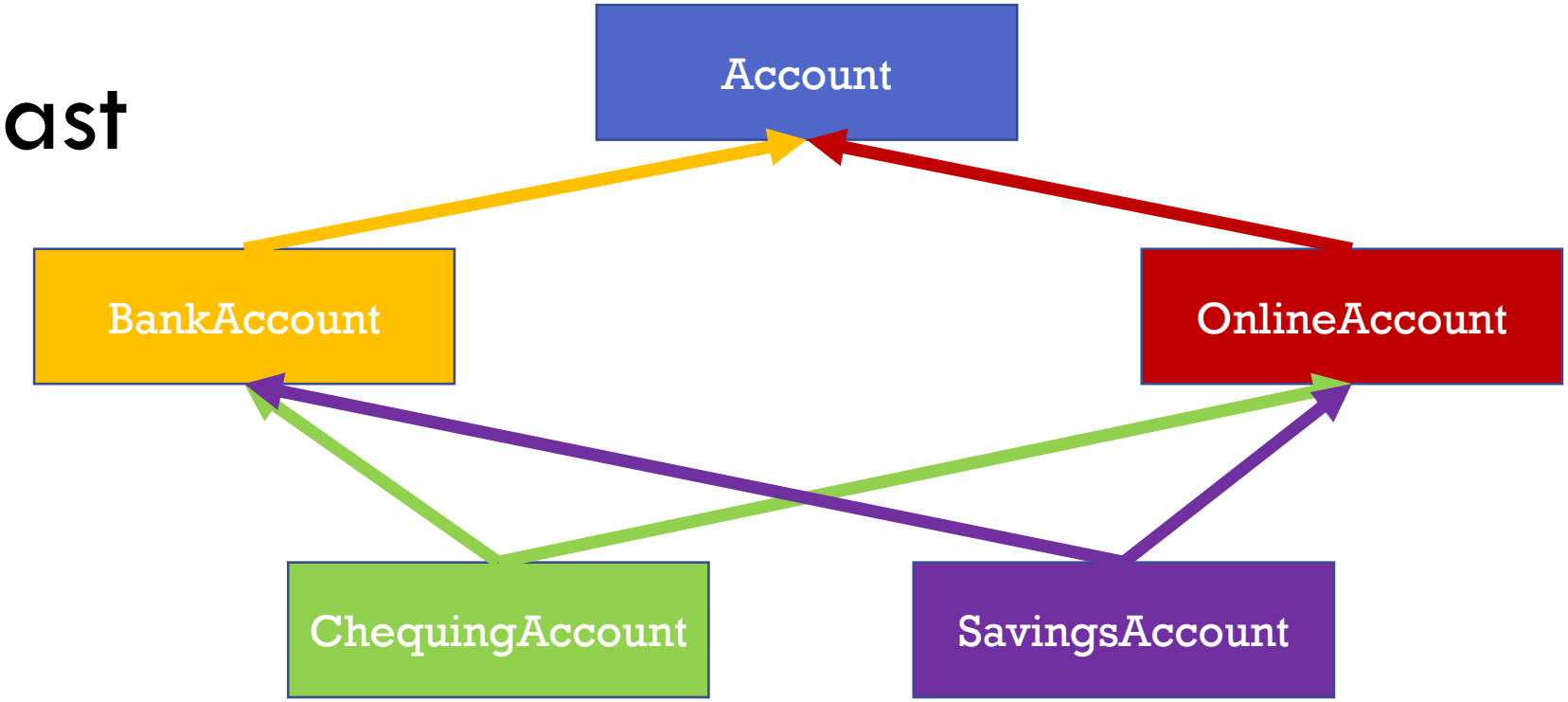


Downcast



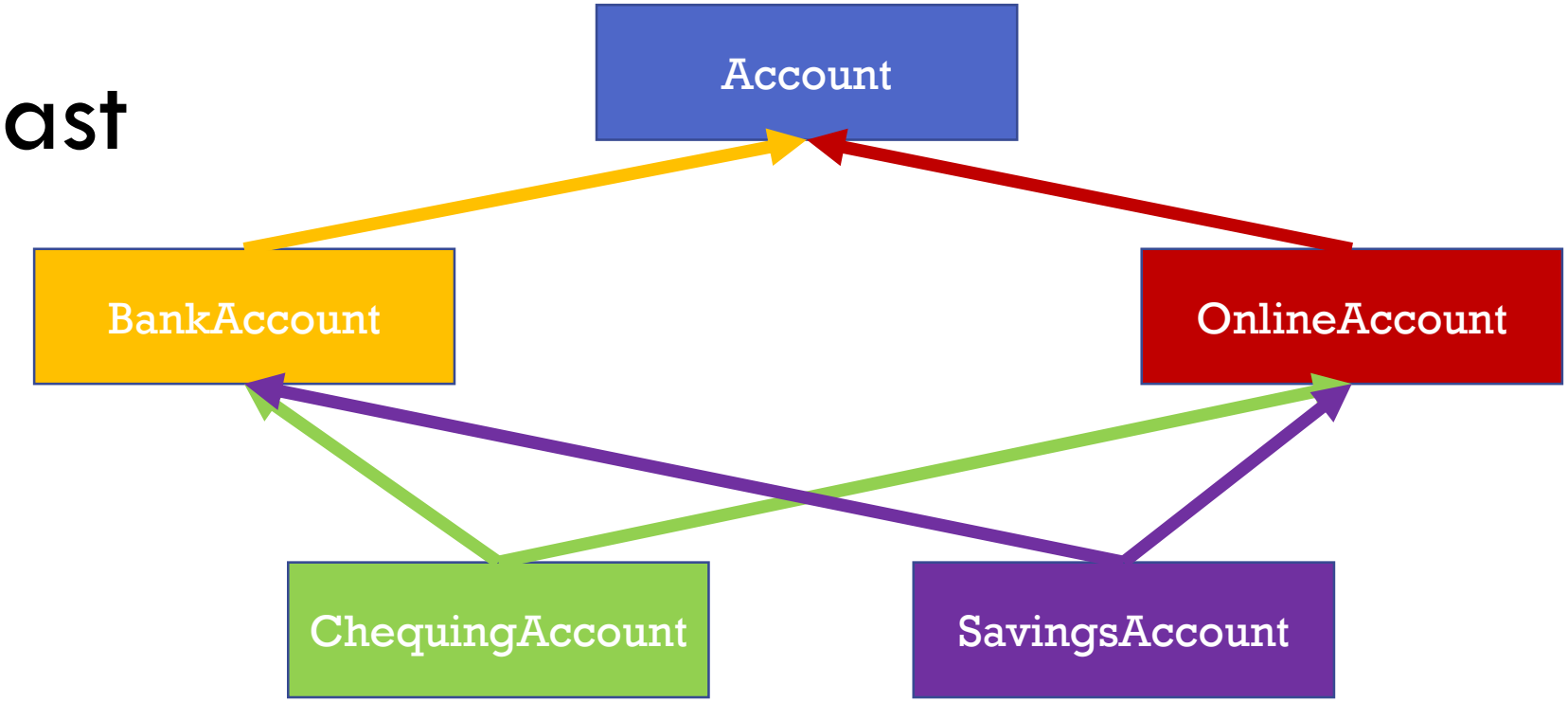
- SavingsAccount s;
- OnlineAccount * oa = dynamic_cast<OnlineAccount *>(&s); // Pointer upcast
- SavingsAccount * sa = dynamic_cast<SavingsAccount *>(oa); // Pointer downcast

Crosscast



- ChequingAccount c;
- BankAccount * ba = dynamic_cast<BankAccount *>(&c); // Pointer upcast

Crosscast



- ChequingAccount c;
- BankAccount * ba = dynamic_cast<BankAccount *>(&c); // Pointer upcast
- OnlineAccount *oa = dynamic_cast<OnlineAccount *>(ba); // Pointer cross cast!

Dynamic cast example (upcast)

account.cpp

```
// Perform an upcast
account_pointer = dynamic_cast<Account *>(&c);

// Do the same thing again to show that
// no cast is required to do an upcast.
account_pointer = &c;
```

(Tangent: check out **typechecking.cpp** for a neat hack.)

Dynamic cast example (downcast/cross cast)

account.cpp

```
// Perform a downcast
sa = dynamic_cast<SavingsAccount *>(oa);

// Perform a cross cast
oa = dynamic_cast<OnlineAccount *>(&c);
```

We can say that the real purpose of dynamic-cast is to allow upcasts within an inheritance hierarchy

Note about dynamic cast

- **Only available when a class is polymorphic**
- Remember that in order to be polymorphic, a class must have at least one virtual function
- **Pro tip: implement a virtual destructor with an empty implementation**

Static cast

- The static_cast operator is used to cast a pointer of one class type to a pointer of another within a inheritance hierarchy
- **Avoids the runtime checks done with dynamic_cast**
- As a result, you can use the static_cast operator with pointers to **nonpolymorphic** types
- **Only valid if the pre-cast type and the post-cast types can be implicitly converted to one another in 1 or both directions**
- You also can use it to carry out some of the conversions performed using C-style casts, generally conversions between related types
- The static_cast operator has the same syntax as dynamic_cast

Const cast

- **Use the const_cast operator to “cast away” the const qualifier**
- It has the same syntax as dynamic_cast
- Between the angle brackets, you specify the same type as the original without the const qualifier
- Using the result is assured to be safe only if the data to which the pointer points was not declared as const or volatile when it was first declared in the program

constcast.cpp

Reinterpret cast

- Converts pointer of one type to pointer of another type, doesn't matter if classes are related
- For inherently risky type casts
- Doesn't let us cast away const
- See **reinterpret.cpp**

```
struct pair { short a, short b );
long value = 0xA224B118;
pair * pd = reinterpret_cast<pair *>(&value);
cout << hex << pd->a;
// This displays the first 2 Bytes of value
```

Casting summary

- Instead of using C-style casting, select the operator that is best suited to each individual purpose
  - Documents the intended reason for the type cast
- Lets the compiler help you
- Watching out for slicing
- Make sure you understand **up_down_cast_example.cpp**

Bottom line: C++ provides safer, better-documented type casts so use them!

ACTIVITY

1. Analyze the up_down_cast_example.cpp file.
2. Can you explain what every single line of code does?
3. Are there any errors or omissions in the comments? Can you improve the comments?

ENUMS

I ♡ enumerations

- Enumerations restrict values to a specific named range of constants
- Underlying type is integral (like Java!)
- C++ enumerations can be **scoped or unscoped**
  - Unscoped are defined with **enum**
  - Scoped are defined with **enum class or enum struct**

I ♡ enumerations

```
const int red = 0;
const int green = 1;
const int blue = 2;
```

vs

```
enum color { red, green, blue}; //same result, less code
```

The C++ enumeration

- A user-defined type that contains a set of named integral constants that are known as enumerators.

- **Unscoped**: enum [identifier] [: type] { enum-list };
  - enum colors : int {red, green, blue}
- **Scoped**: enum **[class | struct]** [identifier] [:type] { enum-list };
  - enum **class** colors : int {red, green, blue}

- Visible in the scope in which they are declared
- Every name in the enum-list is assigned an integral value that corresponds to its place in the order of the enumeration.

Enumerations

We can use **named constants** like this:

```
enum colour { red, green, blue };
colour r = red;
switch(r)
{
case red  : std::cout << "red\n";   break;
case green: std::cout << "green\n"; break;
case blue : std::cout << "blue\n";  break;
}
```

Enumerations

- If not provided, the values of the enumerations begin at 0 (just like Java)
- But in C++ **we can provide the underlying values**:

```
// a = 0, b = 1, c = 10, d = 11, e = 1
// f = 2, g = 12
enum foo
{ a, b, c = 10, d, e = 1, f, g = f + c };
```

Enumerations

- Values can be **converted to integral types**:

```
enum color { red, yellow, green = 20, blue };
color my_colour = red;
int n = blue; // n == 21
```

Enumerations

- Values of integer, floating-point, and other enumeration types can be **converted by static_cast** or explicit cast, to any enumeration type

```
enum status { open = 1, closed = 2, error = 3 };
status file_status = static_cast<status>(3);
std::cout << file_status << std::endl;
```

Enumerations

- We can **omit the name** of the enumeration
- We can ONLY use the enumerators in the enclosing scope

```
// defines a = 0, b = 1, c = 0, d = 2
enum { a, b, c = 0, d = a + 2 };

// prints 0
std::cout << a << std::endl;
```

Enumerations

```
struct X
{
    enum direction { left = 'l', right = 'r' };
};
X x;
X* p = &x;
int a = X::direction::left;
int b = X::left;
int c = x.left;
int d = p->left;
```

What about scoped enumerations?

- So far we've looked at unscoped enumerations..
- There are **no implicit conversions** from the values of a scoped enumerator to integral types
- **static_cast** may be used to obtain the numeric value of the enumerator
- Advantage: **no namespace conflicts**

Scope Example

```
enum Animals {Dog, Cat, Crow};
enum FlyingAnimals {Crow, Sparrow}; //error Crow
already exists

//Compared to scoped enums
enum class Fruit {Apple, Orange, Lemon}; //Lemon
in Fruit scope
enum class YellowFruit {Banana, Lemon}; //Lemon in
YellowFruit scope OK!
```

Example

```
enum class color { red, green = 20, blue };
color r = color::blue;
switch(r){
    case color::red  : cout << "red\n";   break;
    case color::green: cout << "green\n"; break;
    case color::blue : cout << "blue\n";  break;
}
int n = r; // error: no scoped enum to int
conversion
int n = static_cast<int>(r); // OK, n = 21
```

enums.cpp & clothingEnum.cpp

STATIC

Static declarator

- In C++ member variables can be static:
  - Only **one copy** per class exists
  - Permit a single resource to be **shared** between instances
  - **Independent static storage** for life of program
  - This can be useful for the Singleton design pattern (we'll look at it later, but the name is a hint)
  - static **keyword used with declaration** not definition

```
class X { static int n; }; // incomplete declaration
int X::n = 1; // definition
```

Accessing statics

- Two forms can be used:
  1. Qualified name

     Class::member
  2. Member access expression

     Class->member or Class.member

- Exist even if no objects have been defined

Initializing static

- **Can't** be initialized inside the class definition

```
class X {
  static int m = 5; //ERROR
  static int n; //OK

};

int X::n = 5; //OK
```

Static constants: const

- **Can** be initialized with an initializer in which every expression is a constant expression right inside the class definition

```
class X {
  const static int m = 9; //initialized inside
  const static int k;
  static const double * pointer;
};
const int X::k = 3; //initialized outside
const double * X::pointer = new double[3];
```

Static constants: constexpr

- **Must** be initialized with an initializer in which every expression is a constant expression right inside the class definition

```
class X {
  constexpr static int arr[] = { 1, 2, 3 }; // OK
  constexpr static std::complex<double> n
                                 = {1,2}; // OK
  constexpr static int k; // Error
};
```

Static member functions

- Can only access static data and invoke static member functions
- There is no *this pointer
- Cannot be virtual or const