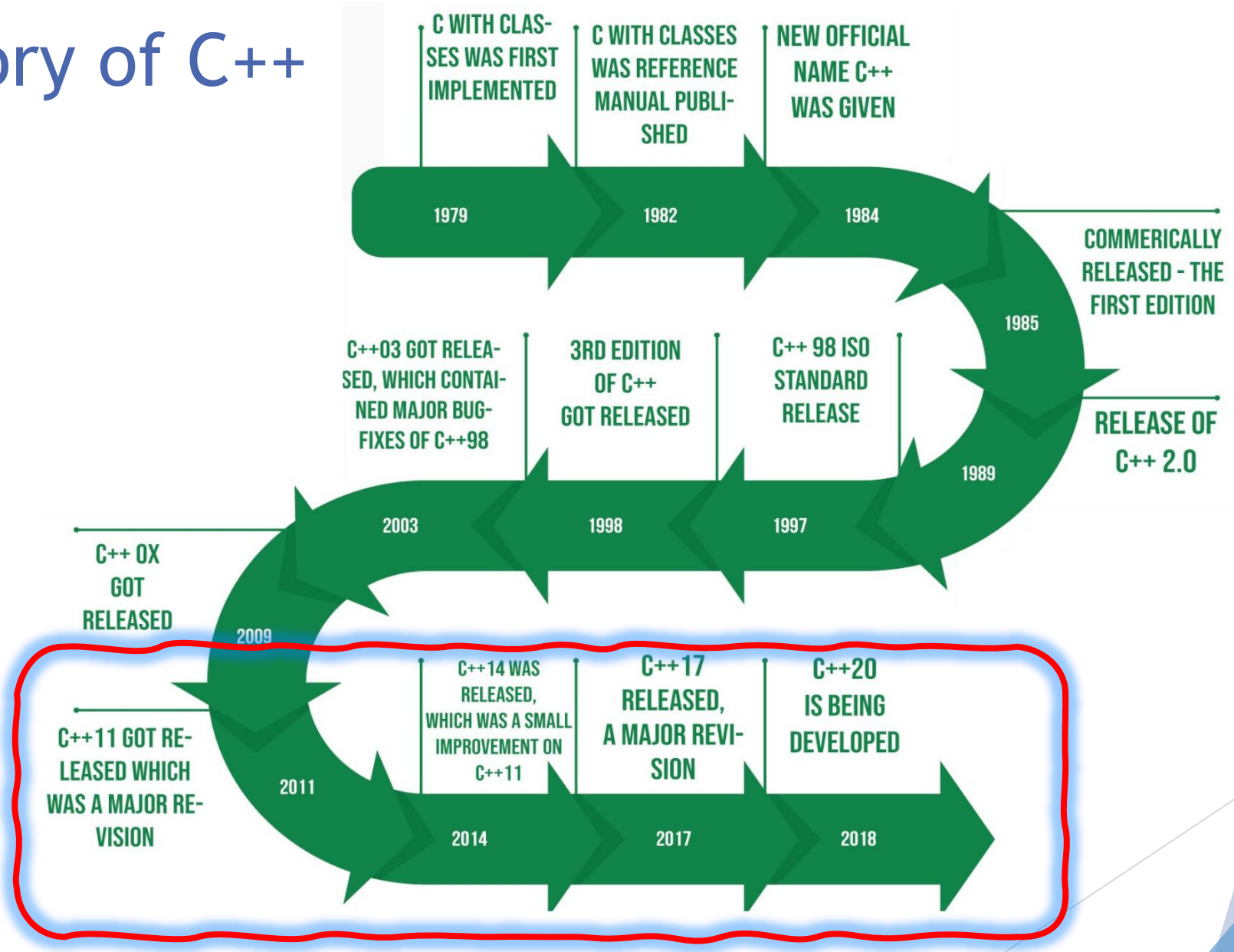
Topics of Modern C++

A Short History

ENNERRE
CONSULTING

History of C++



ENNERRE
CONSULTING

Recall of Advanced Topics

ENNERRE
CONSULTING

Advanced Topics

- Dynamic Memory
- Basic class knowledge
  - Rules for special member function
  - Inheritance
  - Operators
- Standard Type Library

ENNERRE CONSULTING

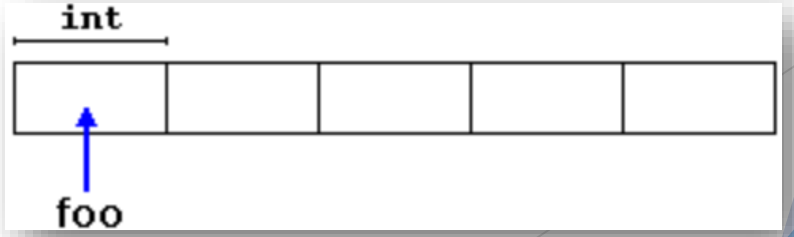
Dynamic Memory

- In many cases programs need to dinamically allocate memory
  - C++ uses language integrated operators
    - `new` and `delete`
  - These operators allocate memory into heap and delete it when it become not necessary
    - The heap becomes also filled with many «holes» depending on subsequent calls of `new` and `delete` operators

Operators new and new[ ]

- Dynamic memory is allocated using operator `new`
  - followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets `[ ]`
  - It returns a pointer to the beginning of the new block of memory allocated
    - Its syntax is:
      - `pointer = new type`
        - used to allocate memory that contain only one single element of type `type`
      - `pointer = new type [number_of_elements]`
        - used to allocate a block (an array) of elements of type type, where `number_of_elements` is an integer value representing the amount of elements

```
1 int * foo;
2 foo = new int [5];
```



ENNERRE CONSULTING

The new operator

- There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new
  - The most important difference is that the size of a regular array needs to be a constant expression, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size
  - The dynamic memory requested by our program is allocated by the system from the memory heap
  - However, computer memory is a limited resource, and it can be exhausted
  - Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system

The new operator

- C++ provides two standard mechanisms to check if the allocation was successful:
  - One is by handling exceptions
    - Using this method, an exception of type `bad_alloc` is thrown when the allocation fails
    - This exception method is the method used by default by new
  - The other method is known as `nothrow`
    - instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a null pointer, and the program continues its execution normally
    - This method can be specified by using a special object called `nothrow`, declared in header `<new>`, as argument for `new`:
      - `foo = new (nothrow) int [5];`

Operators `delete` and `delete[]`

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program
  - once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory
  - This is the purpose of operator delete, whose syntax is:
    - `delete pointer;`
      - releases the memory of a single element allocated using `new`
    - `delete[] pointer;`
      - releases the memory allocated for arrays of elements using new and a size in brackets (`[]`)
- The value passed as argument to delete shall be either a pointer to a memory block previously allocated with `new`, or a null pointer
  - in the case of a null pointer, `delete` produces no effect

The keyword nullptr

- The keyword `nullptr` denotes a predefined **null pointer constant**
- It is a **non-lvalue** of type `std::nullptr_t`
  - `nullptr` can be converted **to a pointer types or bool**, where the result is the **null pointer value of that type or false** respectively

Classes

- Classes are an expanded concept of data structures:
  - like data structures, they can contain data members, but they can also contain functions as members
- An object is an instantiation of a class
  - In terms of variables, a class would be the type, and an object would be the variable
- Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

```
1 class Rectangle {
2     int width, height;
3   public:
4     void set_values (int,int);
5     int area (void);
6 } rect;
```

ENNERRE CONSULTING

Scope Operator

- Scope operator (`::`, `two colons`)
  - used in the definition of members to define a member of a class outside the class itself
  - Complex members is merely declared with its prototype within the class, but its definition is outside it
    - In this outside definition, the operator of scope (`::`) is used to specify that the function being defined is a member of the class and not a regular non-member function
  - The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that
    - in the first case the function is automatically considered an inline member function by the compiler
    - in the second it is a normal (not-inline) class member function
  - This causes no differences in behavior, but only on possible compiler optimizations

Access Modifier

- Default visibility of members is `private`
  - By declaring them `private`, access from outside the class is not allowed
- Others accessors are
  - `protected`
    - Used in inheritence to grant visibility across inheritance hierarchy
  - `public`
    - Used to grant visibility of members freely outside the class

Constructors

- What would happen if we called any member function after variable declaration?
  - An undetermined result, since the members width and height had never been assigned a value
  - In order to avoid that, a class can include a special function called its constructor, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage
  - This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void
  - Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types

```
class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as functional form
- But constructors can also be called with other syntaxes:
  - First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):
    - `class_name object_name = initialization_value;`
- More recently, C++ introduced the possibility of constructors to be called using uniform initialization, which essentially is the same as the functional form, but using braces ({}) instead of parentheses:
  - `class_name object_name { value, value, value, ... }`
- Optionally, this last syntax can include an equal sign before the braces
- Most existing code currently uses functional form, and some newer style guides suggest to choose uniform initialization over the others, even though it also has its potential pitfalls for its preference of *initializer_list* as its type

Uniform Initialization

```cpp
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum() << '\n';
  return 0;
}
```

ENNERRE CONSULTING

Members Initialization in Constructors

- When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members. For example, consider a class with the following declaration:

```
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

- The constructor for this class could be defined, as usual, as:

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

- But it could also be defined using member initialization as:

```
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

- Or even:

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

- Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body
- For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed
- Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor)
  - In these cases, members shall be initialized in the member initialization list

Pointers to Classes

- Objects can also be pointed to by pointers
  - once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer
- Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the `arrow operator` (`->`)

Overloading Operators

- Classes, essentially, define new types to be used in C++ code
  - And types in C++ not only interact with code by means of constructions and assignments, but they also interact by means of operators
  - For a fundamental arithmetic type, the meaning of such operations is generally obvious and unambiguous, but it may not be so for certain class types
- C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes
- Overloadable operators
  - `+  -  *  /`
  - `>  +=  -=  *=  /=`
  - `= >> >>=  ==  !=  >=  ++  --  %  &  ^  !  |`
  - `~  ^=  |=  &&  ||  %=  []  ()  ,`
  - `->*  ->  new delete  new[]  delete[]`

ENNERRE CONSULTING

Overloading Operators

- Operators are overloaded by means of `operator` functions, which are regular functions with special names: their name begins by the operator keyword followed by the operator sign that is overloaded
- The syntax is:

```
type operator sign (parameters) { /*... body ...*/ }
```

Overloading Operators

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

Keyword `this`

- The keyword `this` represents a pointer to the object whose member function is being executed
- It is used within a class's member function to refer to the object itself
- It is also frequently used in `operator=` member functions that return objects by reference

Static Members

- A static data member of a class is also known as a "class variable member“
  - because there is only one common variable for all the objects of that same class, sharing the same value:
    - i.e., its value is not different from one object of this class to another
- Static members have the same properties as non-member variables but they enjoy class scope
  - For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it

```
class Dummy { static int n; }

Dummy::n = 0;
```

- Because it is a common variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
Dummy a; cout << a.n;

cout << Dummy::n;
```

Const Member Functions

- When an object of a class is qualified as a const object:

```
const MyClass myobject;
```

- the access to its data members from outside the class is restricted to read-only, as if all its data members were const for those accessing them from outside the class
- The member functions of a const object can only be called if they are themselves specified as const members;
  - To specify that a member is a const member, the const keyword shall follow the function prototype, after the closing parenthesis for its parameters:

```
int get() const {return x;}
```

Const Member Functions

- Note that `const` can be used to qualify the type returned by a member function
  - This `const` is not the same as the one which specifies a member as `const`
  - Both are independent and are located at different places in the function prototype:

```
int get() const {return x;}          // const member function
const int& get() {return x;}         // member function returning a const&
const int& get() const {return x;} // const member function returning a
                                     // const&
```

Exercise 01

- Create a class that handle the concept of date and time, including validation and date arithmetics (i.e. add of time periods)

Class Templates

- Just like we can create function templates, we can also create class templates, allowing classes to have members that use template parameters as types

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

ENNERRE CONSULTING

Class Templates

- The class that we have just defined serves to store two elements of any valid type
  - For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
1 mypair<int> myobject (115, 36);
```

  - This same class could also be used to create an object to store any other type, such as:

```
1 mypair<double> myfloats (3.0, 2.18);
```

Class Templates

- The constructor is the only member function in the previous class template and it has been defined inline within the class definition itself
- In case that a member function is defined outside the definition of the class template, it shall be preceded with the `template <...>` prefix:

```
1  // class templates
2  #include <iostream>
3  using namespace std;
4
5  template <class T>
6  class mypair {
7      T a, b;
8    public:
9      mypair (T first, T second)
10       {a=first; b=second;}
11     T getmax ();
12 };
13
14 template <class T>
15 T mypair<T>::getmax ()
16 {
17   T retval;
18   retval = a>b? a : b;
19   return retval;
20 }
21
22 int main () {
23   mypair <int> myobject (100, 75);
24   cout << myobject.getmax();
25   return 0;
26 }
```

ENNERRE CONSULTING

Template Specialization

- It is possible to define a different implementation for a template when a specific type is passed as template argument
  - This is called a template specialization
- For example, let's suppose that we have a very simple class called `mycontainer` that can store one element of any type and that has just one member function called increase, which increases its value
- But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type

Template Specialization

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};
```

```
// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

ENNERRE CONSULTING

Exercise 02

- Design a class that handle a circular queue to store any particular type of data

Special Members

- Special member functions are member functions that are implicitly defined as member of classes under certain circumstances
- There are six:

Member function	typical form for class C:
Default constructor	`C::C();`
Destructor	`C::~C();`
Copy constructor	`C::C (const C&);`
Copy assignment	`C& operator= (const C&);`
Move constructor	`C::C (C&&);`
Move assignment	`C& operator= (C&&);`

Default Constructor

- The default constructor is the constructor called when objects of a class are declared, but are not initialized with any arguments
- If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor
- The compiler assumes that Example has a default constructor.
  - Therefore, objects of this class can be constructed by simply declaring them without any arguments

```
1 class Example {
2   public:
3     int total;
4     void accumulate (int x) { total += x; }
5 };
```

But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments

```
1 Example ex;
```

Destructor

- Destructors fulfill the opposite functionality of constructors:
  - They are responsible for the necessary cleanup needed by a class when its lifetime ends
  - The classes we have defined in previous chapters did not allocate any resource and thus did not really require any clean up
- But now, let's imagine that the class in the last example allocates dynamic memory to store the string it had as data member; in this case, it would be very useful to have a function called automatically at the end of the object's life in charge of releasing this memory
  - To do this, we use a destructor
  - A destructor is a member function very similar to a default constructor: it takes no arguments and returns nothing, not even void
  - It also uses the class name as its own name, but preceded with a tilde sign (~)

Destructor

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Copy Constructor

- When an object is passed a named object of its own type as argument, its copy constructor is invoked in order to construct a copy
- A copy constructor is a constructor whose first parameter is of type reference to the class itself (possibly `const` qualified) and which can be invoked with a single argument of this type
  - If a class has no custom `copy` nor `move` constructors (or assignments) defined, an **implicit** copy constructor is provided
- This copy constructor simply performs a copy of its own members
- ThE default copy constructor may suit the needs of many classes, but shallow copies only copy the members of the class themselves, and this is probably not what we expect
  - Because in any circumstances we need to perform a deep copy of object's content

Copy Constructor

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

ENNERRE CONSULTING

Copy Assignment

```
1 MyClass foo;
2 MyClass bar (foo);        // object initialization: copy constructor called
3 MyClass baz = foo;        // object initialization: copy constructor called
4 foo = bar;                // object already initialized: copy assignment called
```

- `baz` is initialized on construction using an equal sign, but this is not an assignment operation!
  - The declaration of an object is not an assignment operation, it is just another of the syntaxes to call single-argument constructors
- The assignment on `foo` is an assignment operation
  - No object is being declared here, but an operation is being performed on an existing object

Copy Assignment

- The copy assignment operator is an overload of operator= which takes a value or reference of the class itself as parameter
- The return value is generally a reference to *this (although this is not required) to use assignment into expressions
  - For example, for a class MyClass, the copy assignment may have the following signature

```
MyClass& operator= (const MyClass&);
```

- The copy assignment operator is also a special function and is also defined implicitly if a class has no custom copy nor move assignments (nor move constructor) defined

Copy Assignment

- But again, the implicit version performs a shallow copy which is suitable for many classes, but not for classes with pointers to objects they handle its storage
  - In this case, not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment.
  - These issues could be solved with a copy assignment that deletes the previous object and performs a deep copy

```
1 Example5& operator= (const Example5& x) {
2   delete ptr;                       // delete currently pointed string
3   ptr = new string (x.content());   // allocate space for new string, and copy
4   return *this;
5 }
6
```

ENNERRE CONSULTING

Move Constructor

- Moving also uses the value of an object to set the value to another object
  - Unlike copying, the content is actually transferred from one object (the source) to the other (the destination):
    - the source loses that content, which is taken over by the destination
- This moving only happens when the source of the value is an unnamed object
  - Unnamed objects are objects that are temporary in nature, and thus haven't even been given a name
    - Typical examples of unnamed objects are return values of functions or type-casts

Move Constructor

- The move constructor and move assignment are members that take a parameter of type rvalue reference to the class itself
  - An rvalue reference is specified by following the type with two ampersands (&&)
  - As a parameter, an rvalue reference matches arguments of temporaries of this type.
- The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete
  - In such objects, copying and moving are really different operations:
    - Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B
    - Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer

Implicit Members

- The six special members functions described above are members implicitly declared on classes under certain circumstances:

MEMBER FUNCTION	IMPLICITLY DEFINED:	DEFAULT DEFINITION:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

Implicit Members

- Not all special member functions are implicitly defined in the same cases
  - This is mostly due to backwards compatibility with C structures and earlier C++ versions, and in fact some include deprecated cases
  - Fortunately, each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords default and delete, respectively
- The syntax is either one of:

```
function_declaration = default;
function_declaration = delete;
```

Friendship

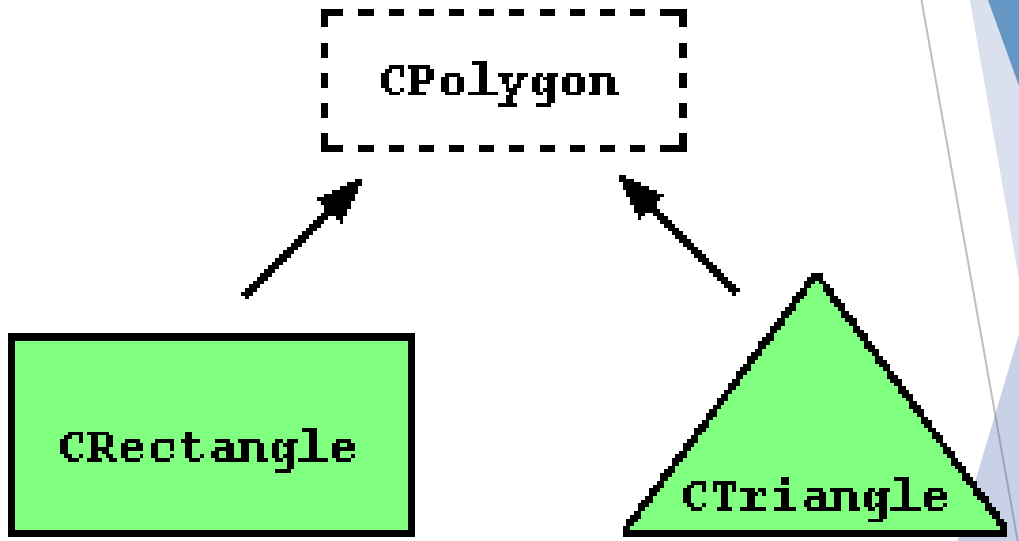
- Private and protected members of a class cannot be accessed from outside the same class in which they are declared
  - However, this rule does not apply to "friends“
    - Friends are functions or classes declared with the friend keyword
    - A non-member function can access the private and protected members of a class if it is declared a `friend` of that class

Exercise 03

- Handle arithmetics fractions and simple math operations

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class
  - This process, known as inheritance, involves a *base class* and a *derived class*
    - The derived class inherits the members of the base class, on top of which it can add its own members
- Let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles
  - These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a *height* and a *width* (or base).
  - This could be represented in the world of classes with a `class Polygon` from which we would derive the two other ones: `Rectangle` and `Triangle`



The Polygon class would contain members that are common for both types of polygon. In our case: `width` and `height`. And `Rectangle` and `Triangle` would be its derived classes, with specific features that are different from one type of polygon to the other.

Inheritance

- The inheritance relationship of two classes is declared in the derived class
- Derived classes definitions use the following syntax:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

- Where `derived_class_name` is the name of the derived class and `base_class_name` is the name of the class on which it is based
- The public access specifier may be replaced by any one of the other access specifiers (protected or private)
  - This access specifier limits the most accessible level for the members inherited from the base class
    - The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

ENNERRE CONSULTING

Inheritance

- The protected access specifier used in class Polygon is similar to private
  - Its only difference occurs in fact with inheritance:
    - When a class inherits another one, the members of the derived class can access the protected members inherited from the base class, but not its private members
- By declaring width and height as protected instead of private, these members are also accessible from the derived classes Rectangle and Triangle, instead of just from members of Polygon
  - If they were public, they could be accessed just from anywhere
- We can summarize the different access types according to which functions can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

What is inherited from the base class?

- In principle, a publicly derived class inherits access to every member of a base class except:
  - its constructors and its destructor
  - its assignment operator members (operator=)
  - its friends
  - its private members
- Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class
- Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments)
  - Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:

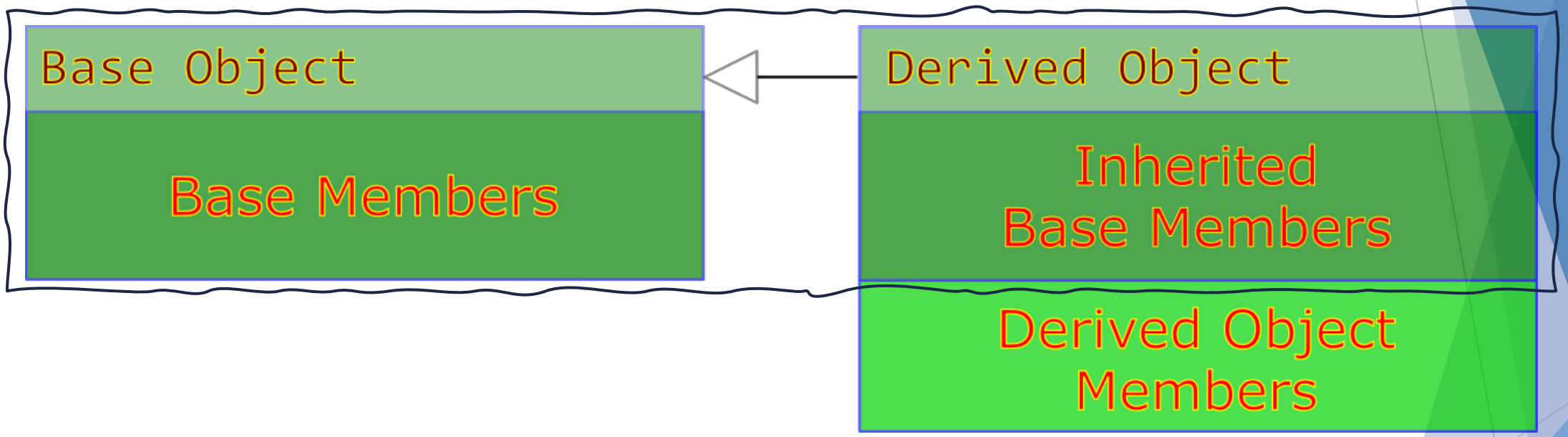
```
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
```

Multiple Inheritance

- A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon)
  - For example, if the program had a specific class to print on screen called Output, and we wanted our classes Rectangle and Triangle to also inherit its members in addition to those of Polygon we could write:

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

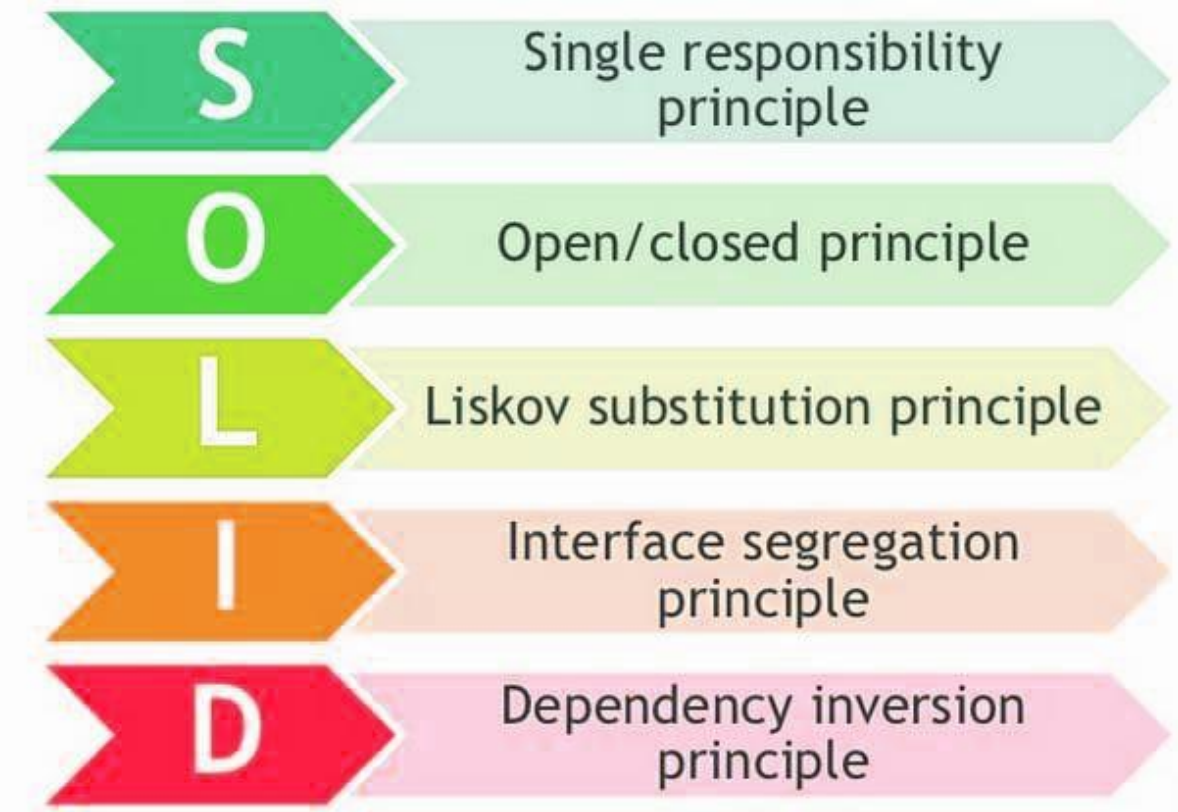
Slicing


Base Object
Base Members
Derived Object
Inherited
Base Members
Derived Object
Members

Polymorphism

- One of the key features of class inheritance is that a pointer (or reference) to a derived class is type-compatible with a pointer to its base class
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature
- A `virtual` member is a member function that can be redefined in a derived class, while preserving its calling properties through references
- Abstract base classes are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions)
  - The syntax is to replace their definition by `=0` (an equal sign and a zero)

SOLID Principles



ENNERRE CONSULTING

SOLID Principles

- Single Responsibility
  - A class should have one and only one reason to change, meaning that a class should have only one job
- Open-Closed
  - Objects or entities should be open for extension but closed for modification
- Liskov Substitution
  - Let q(x) be a property provable about objects of x of type T; then q(y) should be provable for objects y of type S where S is a subtype of T
- Interface Segregation
  - A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use
- Dependency Inversion
  - Entities must depend on abstractions, not on concretions; it states that the high-level module must not depend on the low-level module, but they should depend on abstractions

Future Readings

- Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm (1994):

Design Patterns: Elements of Reusable Object-Oriented Software

- The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly. Due to its lengthy name, people started to call it “the book by the gang of four” which was soon shortened to simply “the GoF book”

Exercise 04

- In a canvas we must draw multiples shapes.

Language Features

ENNERRE
CONSULTING

Language Features

- `noexcept` keyword
- Type inference and `auto` declaration
- Range-based for loops
- Inline namespaces
- Scoped enumerations
- Alias declarations and alias templates
- String views
- Concepts
- Lambdas and functions objects
- Move semantics
  - Motivation and effects
  - x-values categories
  - `move()` and `forward<>()`
    - Universal forwarding references

ENNERRE CONSULTING

The operator noexcept

- The noexcept operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions
  - It can be used within a function template's noexcept specifier to declare that the function will throw exceptions for some types but not others

```cpp
void may_throw();
void no_throw() noexcept;
auto lmay_throw = []{};
auto lno_throw = []() noexcept {};

class T
{
public:
    ~T(){} // dtor prevents move ctor
           // copy ctor is noexcept
};

class U
{
public:
    ~U(){} // dtor prevents move ctor
           // copy ctor is noexcept(false)
    std::vector<int> v;
};

class V
{
public:
    std::vector<int> v;
};

int main()
{
    T t;
    U u;
    V v;

    std::cout << std::boolalpha <<
        "may_throw() is noexcept(" << noexcept(may_throw()) << ")\n"
        "no_throw() is noexcept(" << noexcept(no_throw()) << ")\n"
        "lmay_throw() is noexcept(" << noexcept(lmay_throw()) << ")\n"
        "lno_throw() is noexcept(" << noexcept(lno_throw()) << ")\n"
        "~T() is noexcept(" << noexcept(std::declval<T>().~T()) << ")\n"
        // note: the following tests also require that ~T() is noexcept because
        // the expression within noexcept constructs and destroys a temporary
        "T(rvalue T) is noexcept(" << noexcept(T(std::declval<T>())) << ")\n"
        "T(lvalue T) is noexcept(" << noexcept(T(t)) << ")\n"
        "U(rvalue U) is noexcept(" << noexcept(U(std::declval<U>())) << ")\n"
        "U(lvalue U) is noexcept(" << noexcept(U(u)) << ")\n"
        "V(rvalue V) is noexcept(" << noexcept(V(std::declval<V>())) << ")\n"
        "V(lvalue V) is noexcept(" << noexcept(V(v)) << ")\n";
}
```

ENNERRE CONSULTING

Type Inference

- Type Inference refers to automatic deduction of the data type of an expression in a programming language
- Before C++ 11, each data type needed to be explicitly declared at compile-time
- In new C++ keyword auto and decltype are included which allows a programmer to leave the type deduction to the compiler itself
  - With type inference capabilities, we can spend less time having to write out things the compiler already knows
  - As all the types are deduced in the compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program

The keyword `auto`

- The `auto` keyword in C++ specifies that the type of the variable that is being declared will be automatically deducted from its initializer
  - In the case of functions, if their return type is `auto` then that will be evaluated by return type expression at runtime
- Good use of `auto` is to avoid long initializations when creating iterators for containers
- The variable declared with `auto` keyword should be initialized at the time of its declaration only or else there will be a compile-time error

The Operator `typeid()`

- `typeid` is an operator that is used where the dynamic type of an object needs to be known
  - `typeid(x).name()` returns the data type of `x`, for example, it returns:
    - `'i'` for integers, `'d'` for doubles,
    - `'f'` for float, `'c'` for char,
    - `'Pi'` for the pointer to the integer,
    - `'Pd'` for the pointer to double,
    - `'Pf'` for the pointer to float,
    - `'Pc'` for the pointer to char,
    - `'PPi'` for the pointer to pointer to integer.
    - Single Pointer results in prefix `'P'`,
    - double-pointer results in `'PP'` as a prefix and so on.
  - But the actual name returned is mostly compiler-dependent

Type Inference with auto

```
auto x = 4;
auto y = 3.37;
  auto z = 3.37f;
  auto c = 'a';
auto ptr = &x;
  auto pptr = &ptr; //pointer to a pointer
cout << typeid(x).name() << endl
     << typeid(y).name() << endl
     << typeid(z).name() << endl
     << typeid(c).name() << endl
     << typeid(ptr).name() << endl
       << typeid(pptr).name() << endl;
```

output

```
i
d
f
c
Pi
PPi
```

ENNERRE CONSULTING

Range Based Loops

- Executes a for loop over a range
  - Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container

Syntax

attr(optional) **for** **(** *init-statement*(optional) *range-declaration* **:** *range-expression* **)**
loop-statement

attr - any number of attributes

init-statement - (since C++20) either

- an expression statement (which may be a *null statement* ";")
- a simple declaration, typically a declaration of a variable with initializer, but it may declare arbitrarily many variables or be a structured binding declaration
- an alias declaration (since C++23)

Note that any *init-statement* must end with a semicolon **;**, which is why it is often described informally as an expression or a declaration followed by a semicolon.

range-declaration - a declaration of a named variable, whose type is the type of the element of the sequence represented by *range-expression*, or a reference to that type. Often uses the auto specifier for automatic type deduction

range-expression - any expression that represents a suitable sequence (either an array or an object for which begin and end member functions or free functions are defined, see below) or a braced-init-list.

loop-statement - any statement, typically a compound statement, which is the body of the loop

Range Base Loop Sample

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    auto v = vector<int>{ 32,3456,478,12543,54867,23564,5987,3456,5896 };
    cout << "Items in vector incremented by 1" << endl;
    for (auto c : v)
        cout << ++c << endl;
    cout << "Items as reference in vector incremented by 1" << endl;
    for (auto& c : v)
        cout << ++c << endl;
    cout << "Items in vector after increment" << endl;
    for (auto c : v)
        cout << c << endl;
}
```

```
Items in vector incremented by 1
33
3457
479
12544
54868
23565
5988
3457
5897
Items as reference in vector incremented by 1
33
3457
479
12544
54868
23565
5988
3457
5897
Items in vector after increment
33
3457
479
12544
54868
23565
5988
3457
5897
```

ENNERRE CONSULTING

Structured Binding Declaration

- Binds the specified names to sub-objects or elements of the initializer
  - Like a reference, a structured binding is an alias to an existing object
  - Unlike a reference, a structured binding does not have to be of a reference type
- Binding an array
- Binding a tuple-like type

```
int a[2] = {1, 2};

auto [x, y] = a;     // creates e[2], copies a into e,
                     // then x refers to e[0], y refers to e[1]
auto& [xr, yr] = a; // xr refers to a[0], yr refers to a[1]
```

```
float x{};
char  y{};
int   z{};

std::tuple<float&, char&&, int> tpl(x, std::move(y), z);
const auto& [a, b, c] = tpl;
```

Namespaces

- Namespaces provide a method for preventing name conflicts in large projects
  - Entities declared inside a namespace block are placed in a namespace scope, which prevents them from being mistaken for identically-named entities in other scopes
  - Entities declared outside all namespace blocks belong to the global namespace
  - The global namespace belongs to the global scope, and can be referred to explicitly with a leading `::`
    - While it has no declaration, the global namespace is not an unnamed namespace.
- Multiple namespace blocks with the same name are allowed
  - All declarations within these blocks are declared in the same namespace scope

Namespaces

- Namespace definitions are only allowed at namespace scope, including the global scope
- To reopen an existing namespace (formally, to be an extension-namespace-definition), the lookup for the identifier used in the namespace definition must resolve to a namespace name (not a namespace alias), that was declared as a member of the enclosing namespace or of an inline namespace within an enclosing namespace
- The namespace-body defines a namespace scope, which affects name lookup
- All names introduced by the declarations that appear within namespace-body (including nested namespace definitions) become members of the namespace identifier, whether this namespace definition is the original namespace definition (which introduced identifier), or an extension namespace definition (which "reopened" the already defined namespace)
- A namespace member that was declared within a namespace body may be defined or redeclared outside of it using explicit qualification

Namespaces

```
namespace Q
{
    namespace V   // V is a member of Q, and is fully defined within Q
    { // namespace Q::V { // C++17 alternative to the lines above
        class C { void m(); }; // C is a member of V and is fully defined within V
                               // C::m is only declared
        void f(); // f is a member of V, but is only declared here
    }

    void V::f() // definition of V's member f outside of V
                // f's enclosing namespaces are still the global namespace, Q, and Q::V
    {
        extern void h(); // This declares ::Q::V::h
    }

    void V::C::m() // definition of V::C::m outside of the namespace (and the class body)
                   // enclosing namespaces are the global namespace, Q, and Q::V
    {}
}
```

Namespaces

```
namespace Q
{
    namespace V    // original-namespace-definition for V
    {
        void f();  // declaration of Q::V::f
    }

    void V::f() {} // OK
    void V::g() {} // Error: g() is not yet a member of V

    namespace V    // extension-namespace-definition for V
    {
        void g();  // declaration of Q::V::g
    }
}

namespace R           // not an enclosing namespace for Q
{
    void Q::V::g() {} // Error: cannot define Q::V::g inside R
}

void Q::V::g() {}     // OK: global namespace encloses Q
```

Namespaces

- Names introduced by `friend` declarations within a non-local `class X` become members of the innermost enclosing namespace of `X`, but they do not become visible to ordinary name lookup (neither unqualified nor qualified) unless a matching declaration is provided at namespace scope, either before or after the class definition
- Only the innermost enclosing namespace is considered by such friend declaration when deciding whether the name would conflict with a previously declared name

Namespaces

```
void h(int);
namespace A
{
    class X
    {
        friend void f(X);       // A::f is a friend

        class Y
        {
            friend void g();    // A::g is a friend
            friend void h(int); // A::h is a friend, no conflict with ::h
        };
    };
    // A::f, A::g and A::h are not visible at namespace scope
    // even though they are members of the namespace A

    X x;
    void g()  // definition of A::g
    {
        f(x); // A::X::f is found through ADL
    }

    void f(X) {}   // definition of A::f
    void h(int) {} // definition of A::h
    // A::f, A::g and A::h are now visible at namespace scope
    // and they are also friends of A::X and A::X::Y
}
```

Inline Namespaces

- An inline namespace is a namespace that uses the optional keyword `inline` in its original-namespace-definition
  - Members of an inline namespace are treated as if they are members of the enclosing namespace in many situations
  - This property is transitive: if a namespace N contains an inline namespace M, which in turn contains an inline namespace O, then the members of O can be used as though they were members of M or N
- A using-directive that names the inline namespace is implicitly inserted in the enclosing namespace (similar to the implicit using-directive for the unnamed namespace)
- In argument-dependent lookup, when a namespace is added to the set of associated namespaces, its inline namespaces are added as well, and if an inline namespace is added to the list of associated namespaces, its enclosing namespace is added as well
- Each member of an inline namespace can be partially specialized, explicitly instantiated, or explicitly specialized as if it were a member of the enclosing namespace
- Qualified name lookup that examines the enclosing namespace will include the names from the inline namespaces even if the same name is present in the enclosing namespace

Inline Namespaces

```
// in C++14, std::literals and its member namespaces are inline
{
    using namespace std::string_literals; // makes visible operator""s
                                          // from std::literals::string_literals
    auto str = "abc"s;
}

{
    using namespace std::literals; // makes visible both
                                   // std::literals::string_literals::operator""s
                                   // and std::literals::chrono_literals::operator""s
    auto str = "abc"s;
    auto min = 60s;
}

{
    using std::operator""s; // makes both std::literals::string_literals::operator""s
                            // and std::literals::chrono_literals::operator""s visible
    auto str = "abc"s;
    auto min = 60s;
}
```

Unnamed Namespaces

- The unnamed-namespace-definition is a namespace definition of the form

```
inline(optional) namespace attr (optional) { namespace-body}
```

- This definition is treated as a definition of a namespace with unique name and a using-directive in the current scope that nominates this unnamed namespace
  - Note: implicitly added using directive makes namespace available for the qualified name lookup and unqualified name lookup, but not for the argument-dependent lookup
- The unique name is unique over the entire program, but within a translation unit each unnamed namespace definition maps to the same unique name:
  - multiple unnamed namespace definitions in the same scope denote the same unnamed namespace

Unnamed Namespaces

```
namespace
{
    int i; // defines ::(unique)::i
}

void f()
{
    i++;   // increments ::(unique)::i
}

namespace A
{
    namespace
    {
        int i;        // A::(unique)::i
        int j;        // A::(unique)::j
    }

    void g() { i++; } // A::(unique)::i++
}

using namespace A; // introduces all names from A into global namespace

void h()
{
    i++;    // error: ::(unique)::i and ::A::(unique)::i are both in scope
    A::i++; // ok, increments ::A::(unique)::i
    j++;    // ok, increments ::A::(unique)::j
}
```

`using` Declarations

- Introduces a name that is defined elsewhere into the declarative region where this using-declaration appears.
- Until C++17:

```
using typename(optional) nested-name-specifier unqualified-id;
```

- Since C++17:

```
using declarator-list;
```

  - A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator.

- Since C++20:
  - Using-declarations can be used to introduce namespace members into other namespaces and block scopes, or to introduce base class members into derived class definitions, or to introduce enumerators into namespaces, block, and class scopes

using Declarations

```
void f();
namespace A
{
    void g();
}

namespace X
{
    using ::f;         // global f is now visible as ::X::f
    using A::g;        // A::g is now visible as ::X::g
    using A::g, A::g; // (C++17) OK: double declaration allowed at namespace scope
}

void h()
{
    X::f(); // calls ::f
    X::g(); // calls A::g
}
```

ENNERRE CONSULTING

`using` Declarations

- If, after the using-declaration was used to take a member from a namespace, the namespace is extended and additional declarations for the same name are introduced,
  - those additional declarations do not become visible through the using-declaration (in contrast with using-directive)
  - One exception is when a using-declaration names a class template:
    - partial specializations introduced later are effectively visible, because their lookup proceeds through the primary template

using Declarations

```
namespace A
{
    void f(int);
}
using A::f; // ::f is now a synonym for A::f(int)

namespace A       // namespace extension
{
    void f(char); // does not change what ::f means
}

void foo()
{
    f('a'); // calls f(int), even though f(char) exists.
}

void bar()
{
    using A::f; // this f is a synonym for both A::f(int) and A::f(char)
    f('a');     // calls f(char)
}
```

`using` Directive

- A using-directive is a block-declaration with the following syntax:

```
using namespace nested-name-specifier (optional) namespace-name;
```

- Using-directives are allowed only in namespace scope and in block scope
  - From the point of view of unqualified name lookup of any name after a using-directive and until the end of the scope in which it appears, every name from namespace-name is visible as if it were declared in the nearest enclosing namespace which contains both the using-directive and namespace-name
- Using-directive does not add any names to the declarative region in which it appears (unlike the using-declaration), and thus does not prevent identical names from being declared
- Using-directives are transitive for the purposes of unqualified lookup:
  - if a scope contains a using-directive that nominates a namespace-name, which itself contains using-directive for some namespace-name-2, the effect is as if the using directives from the second namespace appear within the first
  - The order in which these transitive namespaces occur does not influence name lookup

using Directive

```
namespace A
{
    int i;
}

namespace B
{
    int i;
    int j;

    namespace C
    {
        namespace D
        {
            using namespace A; // all names from A injected into global namespace

            int j;
            int k;
            int a = i;         // i is B::i, because A::i is hidden by B::i
        }

        using namespace D; // names from D are injected into C
                           // names from A are injected into global namespace

        int k = 89; // OK to declare name identical to one introduced by a using
        int l = k;  // ambiguous: C::k or D::k
        int m = i;  // ok: B::i hides A::i
        int n = j;  // ok: D::j hides B::j
    }
}
```

using Directive

```
namespace D
{
    int d1;
    void f(char);
}
using namespace D; // introduces D::d1, D::f, D::d2, D::f,
                   // E::e, and E::f into global namespace!

int d1;            // OK: no conflict with D::d1 when declaring

namespace E
{
    int e;
    void f(int);
}

namespace D            // namespace extension
{
    int d2;
    using namespace E; // transitive using-directive
    void f(int);
}

void f()
{
    d1++;    // error: ambiguous ::d1 or D::d1?
    ::d1++;  // OK
    D::d1++; // OK
    d2++;    // OK, d2 is D::d2

    e++;     // OK: e is E::e due to transitive using

    f(1);    // error: ambiguous: D::f(int) or E::f(int)?
    f('a');  // OK: the only f(char) is D::f(char)
}
```

Enumerations

- An enumeration is a distinct type whose value is restricted to a range of values, which may include several explicitly named constants ("enumerators").
- The values of the constants are values of an integral type known as the underlying type of the enumeration
- An enumeration has the same size, value representation, and alignment requirements as its underlying type
  - Furthermore, each value of an enumeration has the same representation as the corresponding value of the underlying type

Unscoped Enumerations

```
enum name(optional) { enumerator = constexpr , enumerator = constexpr , ... }
```

```
enum name(optional) : type { enumerator = constexpr , enumerator = constexpr , ... }
```

```
enum name : type ;
```

- First declares an unscoped enumeration type whose underlying type is not fixed (in this case, the underlying type is an implementation-defined integral type that can represent all enumerator values
  - this type is not larger than int unless the value of an enumerator cannot fit in an int or unsigned int
  - If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0
  - If no integral type can represent all the enumerator values, the enumeration is ill-formed).
- Second declares an unscoped enumeration type whose underlying type is fixed
- Opaque enum declaration (third) for an unscoped enumeration must specify the name and the underlying type
- Each enumerator becomes a named constant of the enumeration's type (that is, name), visible in the enclosing scope, and can be used whenever constants are required

Unscoped Enumerations

```
enum Color { red, green, blue };
Color 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;
}
```

ENNERRE CONSULTING

Scoped Enumerations

```
enum struct|class name { enumerator = constexpr , enumerator = constexpr , ... }
```

```
enum struct|class name : type { enumerator = constexpr , enumerator = constexpr , ... }
```

```
enum struct|class name ;
```

```
enum struct|class name : type ;
```

1. declares a scoped enumeration type whose underlying type is `int` (the keywords `class` and `struct` are exactly equivalent)
2. declares a scoped enumeration type whose underlying type is `type`
3. opaque enum declaration for a scoped enumeration whose underlying type is `int`
4. opaque enum declaration for a scoped enumeration whose underlying type is `type`

- Each enumerator becomes a named constant of the enumeration's type (that is, name), which is contained within the scope of the enumeration, and can be accessed using scope resolution operator
- There are no implicit conversions from the values of a scoped enumerator to integral types, although `static_cast` may be used to obtain the numeric value of the enumerator

Scoped Enumerations

- An enumeration can be initialized from an integer without a cast, using list initialization, if **all of** the following are true:
  - The initialization is direct-list-initialization
  - The initializer list has only a single element
  - The enumeration is either scoped or unscoped with underlying type fixed
  - The conversion is non-narrowing
- This makes it possible to introduce new integer types that enjoy the same existing calling conventions as their underlying integer types, even on ABIs that penalize passing/returning structures by value

Using-enum-declaration

- A using-enum-declaration introduces the enumerator names of the named enumeration as if by a using-declaration for each enumerator
- When in class scope, a using-enum-declaration adds the enumerators of the named enumeration as members to the scope, making them accessible for member lookup

```
enum class fruit { orange, apple };

struct S
{
    using enum fruit;
};

void f()
{
    S s;
    s.orange;
    S::orange;
}
```

ENNERRE CONSULTING

Type and Templates Alias

- Type alias is a name that refers to a previously defined type (similar to typedef)
- Alias template is a name that refers to a family of types

```
using identifier attr(optional) = type-id ;
```

```
template < template-parameter-list >
using identifier attr(optional) = type-id ;
```

ENNERRE CONSULTING

Type and Templates Alias

- A type alias declaration introduces a name which can be used as a synonym for the type denoted by type-id
  - It does not introduce a new type and it cannot change the meaning of an existing type name
  - There is no difference between a type alias declaration and typedef declaration
  - This declaration may appear in block scope, class scope, or namespace scope
- An alias template is a template which, when specialized, is equivalent to the result of substituting the template arguments of the alias template for the template parameters in the type-id

```
template<class T>
struct Alloc {};

template<class T>
using Vec = vector<T, Alloc<T>>; // type-id is vector<T, Alloc<T>>

Vec<int> v; // Vec<int> is the same as vector<int, Alloc<int>>
```

String View

- String_view is literally just a view to the original data or a view of a string
  - string_view, is a window, which is only a view of the string and cannot be used to modify the actual string
  - it allows to point into an existing string at some offset

ENNERRE CONSULTING

String View

- Internally, string_view object has a pointer and a size
  - it sets the pointer to the beginning of the original string and sets its size to the size of the original string
    - that way it avoids the heap allocation
  - Any changes made in the original `string text` would reflect in `string_view strV` but would not reflect in the `string str`

```
#include <iostream>
#include <string>
#include <string_view>

int main(){
    char text[] =  "example";
    std::string str = text;
    std::string_view strV = text;
}
```

ENNERRE CONSULTING

String View

- std::string_view contains functions that let us manipulate the view of the string

Element access	
operator[]	accesses the specified character
at	accesses the specified character with bounds checking
front	accesses the first character
back	accesses the last character
data	returns a pointer to the first character of a view
Capacity	
sizelength	returns the number of characters
max_size	returns the maximum number of characters
empty	checks whether the view is empty
Modifiers	
remove_prefix	shrinks the view by moving its start forward
remove_suffix	shrinks the view by moving its end backward
Swap	swaps the contents

Operations	
copy	copies characters
substr	returns a substring
compare	compares two views
starts_with	checks if the string view starts with the given prefix
ends_with	checks if the string view ends with the given suffix
contains	checks if the string view contains the given substring or character
find	find characters in the view
rfind	find the last occurrence of a substring
find_first_of	find first occurrence of characters
find_last_of	find last occurrence of characters
find_first_not_of)	find first absence of characters
find_last_not_of	find last absence of characters

Concepts

- The concepts library provides definitions of fundamental library concepts that can be used to perform compile-time validation of template arguments and perform function dispatch based on properties of types
  - These concepts provide a foundation for equational reasoning in programs
- Most concepts in the standard library impose both syntactic and semantic requirements
  - It is said that a standard concept is satisfied if its syntactic requirements are met, and is modeled if it is satisfied and its semantic requirements (if any) are also met
- In general, only the syntactic requirements can be checked by the compiler
  - If the validity or meaning of a program depends whether a sequence of template arguments models a concept, and the concept is satisfied but not modeled, or if a semantic requirement is not met at the point of use, the program is ill-formed, no diagnostic required

Lambda and Function Objects

ENNERRE
CONSULTING

Lambda Expressions

- C++ 11 introduced lambda expressions to allow inline functions which can be used for short snippets of code that are not going to be reused and therefore do not require a name
- In their simplest form a lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

Lambda Expressions

- **capture clause**
  - a comma-separated list of zero or more captures, optionally beginning with a capture-default
  - A lambda expression can use a variable without capturing it if the variable
    - is a non-local variable or has static or thread local storage duration (in which case the variable cannot be captured)
    - is a reference that has been initialized with a constant expression
  - A lambda expression can read the value of a variable without capturing it if the variable
    - has `const` non-volatile integral or enumeration type and has been initialized with a constant expression
    - is `constexpr` and has no mutable members

Lambda Expressions

- **parameters**
  - a non-empty comma-separated list of template parameters, used to provide names to the template parameters of a generic lambda
- **return type**
  - Is optional (detected from type inference by result) and specify the return type of lambda
- **method definition**
  - The body of expression

Lambda Expressions

- Lambda capture
  - The captures is a comma-separated list of zero or more captures, optionally beginning with the capture-default
    - The capture list defines the outside variables that are accessible from within the lambda function body
    - The only capture defaults are
      - `&` implicitly capture the used variables with automatic storage duration by reference
      - `=` implicitly capture the used variables with automatic storage duration by copy
    - The current object (`*this`) can be implicitly captured if either capture default is present
    - If implicitly captured, it is always captured by reference, even if the capture default is `=`
    - The implicit capture of `*this` when the capture default is = is deprecated since C++20

Lambda Expressions

```cpp
struct S2 { void f(int i); };
void S2::f(int i)
{
    [&] {};          // OK: by-reference capture default
    [&, i] {};       // OK: by-reference capture, except i is captured by copy
    [&, &i] {};      // Error: by-reference capture when by-reference is the default
    [&, this] {};    // OK, equivalent to [&]
    [&, this, i] {}; // OK, equivalent to [&, i]
}
```

```cpp
struct S2 { void f(int i); };
void S2::f(int i)
{
    [=] {};          // OK: by-copy capture default
    [=, &i] {};      // OK: by-copy capture, except i is captured by reference
    [=, *this] {}; // until C++17: Error: invalid syntax
                   // since C++17: OK: captures the enclosing S2 by copy
    [=, this] {};  // until C++20: Error: this when = is the default
                   // since C++20: OK, same as [=]
}
```

```cpp
struct S2 { void f(int i); };
void S2::f(int i)
{
    [i, i] {};         // Error: i repeated
    [this, *this] {}; // Error: "this" repeated (C++17)

    [i] (int i) {};    // Error: parameter and capture have the same name
}
```

ENNERRE CONSULTING

Functors aka Function Objects

- A function object is any object for which the function call operator is defined
  - C++ provides many built-in function objects as well as support for creation and manipulation of new function objects
- C++ defines several function objects that represent common arithmetic and logical operations:
  - Arithmetic operations
  - Comparisons
  - Logical operations
  - Bitwise operations
  - Constrained comparison function objects
- Standard library provides the `<functional>` header to handle functors

Move Semantic

ENNERRE
CONSULTING

Move Semantic

- A **move constructor** is a constructor which can be called with an argument of the same class type and copies the content of the argument, possibly mutating the argument
- The move constructor is typically called when an object is initialized (by direct-initialization or copy-initialization) from **rvalue** of the same type, including
  - initialization: `T a = std::move(b);` or `T a(std::move(b))`;, where `b` is of type `T`;
  - function argument passing: `f(std::move(a))`;, where `a` is of type `T` and `f` is `void f(T t);`
  - function return: `return a;` inside a function such as `T f()`, where `a` is of type `T` which has a move constructor
- Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc.) rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state
  - For example, moving from a `std::string` or from a `std::vector` may result in the argument being left empty
  - However, this behavior should not be relied upon
    - For some types, such as `std::unique_ptr`, the moved-from state is fully specified

Perfect Forwarding

- Perfect forwarding is the act of passing a function's parameters to another function while preserving its reference category
  - It is commonly used by wrapper methods that want to pass their parameters through to another function, often a constructor

Reference Taxonomy

- One of the most misunderstood aspect of C++ is the use of the terms **lvalue** and **rvalue**, and what they mean for how code is interpreted
- Though **lvalue** and **rvalue** are inherited from C, with C++11, this taxonomy was extended and clarified, and 3 more terms were added
  - **glvalue**, **xvalue** and **prvalue**

Reference Taxonomy

- The result of every C++ expression is either an **lvalue**, or an **rvalue**
  - These terms come from C, but the C++ definitions have evolved quite a lot since then, due to the greater expressiveness of the C++ language
- **rvalues** can be split into two subcategories:
  - **xvalues**, and **prvalues**, depending on the details of the expression
    - These subcategories have slightly different properties
      - One of the differences is that **xvalues** can sometimes be treated the same as **lvalues**
  - To cover those cases we have the term **glvalue**
    - if something applies to both **lvalues** and **xvalues** then it is described as applying to **glvalues**

Reference Taxonomy

- **glvalues**
  - A **glvalue** is a **Generalized lvalue**
  - It is used to refer to something that could be either an **lvalue** or an **xvalue**
- **rvalues**
  - The term **rvalue** is inherited from C, where **rvalues** are things that can be on the Right side of an assignment
  - The term **rvalue** can refer to things that are either **xvalues** or **prvalues**
- **lvalues**
  - The term **lvalue** is inherited from C, where **lvalues** are things that can be on the Left side of an assignment
- **xvalues**
  - is an e**X**piring **value**: an unnamed objects that is soon to be destroyed
  - **xvalues** may be either treated as **glvalues** or as **rvalues** depending on context
  - **xvalues** are slightly unusual in that they usually only arise through explicit casts and function calls
  - If an expression is cast to an **rvalue** reference to some type `T` then the result is an **xvalue** of type `T`
    - `static_cast<A&&>(v1)` yields an **xvalue** of type `A`
- **prvalues**
  - A **prvalue** is a **Pure rvalue**; an **rvalue** that is not an **xvalue**
  - Literals other than string literals (which are **lvalues**) are **prvalues**. So `42` is a **prvalue** of type `int`, and `3.141f` is a **prvalue** of type `float`

Reference Taxonomy

- A named object declared with an **rvalue** reference (declared with &&) is also an **lvalue**
  - This is probably the most confusing of the rules, if for no other reason than that it is called an **rvalue reference**
  - The name is just there to indicate that it can bind to an **rvalue**
    - once you've declared a variable and given it a name it's an **lvalue**
    - This is most commonly encountered in function parameters

Reference Binding

- Probably the biggest difference between **lvalues** and **rvalues** is in how they bind to references, though the differences in the type deduction rules can have a big impact too
  - There are two types of references in C++:
    - **lvalue references**, which are declared with a single ampersand, e.g. `T&`
    - **rvalue references** which are declared with a double ampersand, e.g. `T&&`

Reference Binding

- **lvalue references**
  - A **non-const lvalue** reference will only bind to non-const **lvalues** of the same type, or a class derived from the referenced type
  - A **const lvalue reference** on the other hand will also bind to **rvalues**, though again the object bound to the reference must have the same type as the referenced type, or a class derived from the referenced type
    - You can bind both const and non-const values to a const lvalue reference

```
struct C:A{};
int i=42;
A a;
B b;
C c;
const A ca{};

A& r1=a;
A& r2=c;
//A& r3=b; // error, wrong type
int& r4=i;
// int& r5=42; // error, cannot bind rvalue
//A& r6=ca; // error, cannot bind const object to non
const ref
A& r7=r1;
// A& r8=A(); // error, cannot bind rvalue
// A& r9=B().a; // error, cannot bind rvalue
// A& r10=C(); // error, cannot bind rvalue
const A& cr1=a;
const A& cr2=c;
//const A& cr3=b; // error, wrong type
const int& cr4=i;
const int& cr5=42; // rvalue can bind OK
const A& cr6=ca; // OK, can bind const object to const
ref
const A& cr7=cr1;
const A& cr8=A(); // OK, can bind rvalue
const A& cr9=B().a; // OK, can bind rvalue
const A& cr10=C(); // OK, can bind rvalue
const A& cr11=r1;
```

Reference Bindings

- **rvalue references**
  - An **rvalue reference** will only bind to **rvalues** of the same type, or a class derived from the referenced type
  - As for **lvalue references**, the **reference** must be const in order to bind to a **const** object, though **const rvalue** references are much rarer than **const lvalue references**
  - **rvalue references** extend the lifetime of temporary objects in the same way that const **lvalue** references do, so the temporaries associated with rr2, rr5, rr7, rr8, rr9, and rr10 will remain alive until the corresponding references are destroyed

```
const A make_const_A();
// A&& rr1=a; // error, cannot bind lvalue
              //to rvalue reference
A&& rr2=A();
//A&& rr3=B(); // error, wrong type
//int&& rr4=i; // error, cannot bind lvalue
int&& rr5=42;
//A&& rr6=make_const_A(); // error, cannot
    // bind const object to non const ref
const A&& rr7=A();
const A&& rr8=make_const_A();
A&& rr9=B().a;
A&& rr10=C();

// std::move returns an rvalue
A&& rr11=std::move(a);
// A&& rr12=rr11; // error rvalue references
                  // are lvalues
```

A Problem

- If we need to write a function should have the following characteristics:
  - It can take an arbitrary number of arguments
  - Can accept lvalues and rvalues as an argument
  - Forwards its arguments identical to the underlying constructor
- we have a great problem to solve, considering the request and the types of function parameters

A Perfect Factory

- For efficiency reasons, the function template should take its arguments by reference
  - As possible a non-constant lvalue reference
- If we compile the program, we will get a compiler error
  - The reason is that the rvalue – at `create<int>(5)` - can not be bound to a non-constant lvalue reference

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg& a){
  return T(a);
}


int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::cout << std::endl;

}
```

ENNERRE CONSULTING

A Perfect Factory

- Now, we have two ways to solve the issue
  - Change the non-constant lvalue reference in a constant lvalue reference
    - You can bind an rvalue to a constant lvalue reference
    - But that is not perfect because the function argument is constant, and I can not change it
  - Overload the function template for a constant and non-const lvalue reference
    - That is easy and is the right way to go

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg& a){
  return T(a);
}

template <typename T,typename Arg>
T create(const Arg& a){
  return T(a);
}

int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::cout << std::endl;

}
```

ENNERRE CONSULTING

A Perfect Factory

- That was very easy!!!
- The solution has two conceptual issues.
  - To support n different arguments, we have to overload 2^n +1 variations of the function template create. 2^n +1 because the function create without an argument is part of the perfect factory method
  - The function argument mutates in the function body of creating an lvalue, because it has a name
    - Does this matter? Of course, yes. a is not movable anymore.
    - Therefore, I must perform an expensive copy instead of a cheap move
      - But what is even worse? If the T constructor needs an rvalue (i.e. `return T(a)`), it will no longer work

A Perfect Factory

- We have the solution in the shape of the C++ function `std::forward`
  - In this we use the universal reference!!!
    - In form `A&&`
- In this case the template works fine, but it need precisely one argument
  - Wich is perfectly forwarded to the object's constructor

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg&& a){
  return T(std::forward<Arg>(a));
}

int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::cout << std::endl;

}
```

Parameter Pack and Variadic Templates

- A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates)
- A function parameter pack is a function parameter that accepts zero or more function arguments.
- A template with at least one parameter pack is called a variadic template

```
template<class... Types>
struct Tuple {};

Tuple<> t0;           // Types contains no arguments
Tuple<int> t1;        // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<0> t3;          // error: 0 is not a type
```

Pack Expansion

- A pattern followed by an ellipsis, in which the name of at least one parameter pack appears at least once, is expanded into zero or more instantiations of the pattern, where the name of the parameter pack is replaced by each of the elements from the pack, in order
- Instantiations of alignment specifiers are space-separated, other instantiations are comma-separated

```
template<class... Us>
void f(Us... pargs) {}

template<class... Ts>
void g(Ts... args)
{
    f(&args...); // “&args...” is a pack expansion
                 // “&args” is its pattern
}

g(1, 0.2, "a"); // Ts... args expand to int E1, double E2, const char* E3
                // &args... expands to &E1, &E2, &E3
                // Us... pargs expand to int* E1, double* E2, const char** E3
```

Standard Library

ENNERRE
CONSULTING

Standard Library

- Smart Pointers
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
- Effect of new language features on containers
  - `cbegin()`, `emplace()`, …
- New standard containers
  - `std::array<>`
- Multithreading and concurrency
  - `std::thread`, `std::async` and *futures*
  - Mutex, atomics, condition variables

ENNERRE CONSULTING

Smart Pointers

ENNERRE
CONSULTING

Smart Pointers

- Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced)
- unique_ptr
  - make sure that only exactly one copy of an object exists
    - Can be used in the example above for handling dynamically allocated objects in a restricted scope
- shared_ptr
  - is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function
    - This is particularly useful in the context of OOP, to store a pointer as a member variable and return it to access the referenced value outside the scope of the class
- weak_ptr
  - holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`
    - It must be converted to `std::shared_ptr` in order to access the referenced object

ENNERRE CONSULTING

Smart Pointers – unique_ptr

- unique_ptr
  - A unique pointer can be initialized with a pointer upon creation
    - `std::unique_ptr<int> valuePtr(new int(47));`
  - or it can be created without a pointer and assigned one later
    - `std::unique_ptr<int> valuePtr;`
    - `valuePtr.reset(new int(47));`
      - in this case, if the `unique_ptr<>` already holds a pointer to an existing object, this object is deleted first and then the new pointer stored
- The `unique_ptr<>` does not support copying
  - If you try to copy a `unique_ptr<>`, you'll get compiler errors
  - However, it supports move semantics, where the pointer is moved from one `unique_ptr<>` to another, which invalidates the first `unique_ptr<>`

Smart Pointers – unique_ptr

```cpp
void my_func()
{
    int* valuePtr = new int(15);
    int x = 45;
    // ...
    if (x == 45)
        return;   // here we have a memory leak, valuePtr is not deleted
    // ...
    delete valuePtr;
}

int main()
{
}
```

```cpp
#include <memory>

void my_func()
{
    std::unique_ptr<int> valuePtr(new int(15));
    int x = 45;
    // ...
    if (x == 45)
        return;   // no memory leak anymore!
    // ...
}

int main()
{
}
```

ENNERRE CONSULTING

Smart Pointers – shared_ptr

- shared_ptr
  - The shared_pointer is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function
    - This is particularly useful in the context of OOP, to store a pointer as a member variable and return it to access the referenced value outside the scope of the class
  - The usage of smart_ptr allows us to easily pass and return references to objects without running into memory leaks or invalid attempts to access deleted references
  - They are thus a cornerstone of modern memory management

Smart Pointers – shared_ptr

```cpp
#include <memory>

class Foo
{
        public void doSomething();
};

class Bar
{
private:
        std::shared_ptr<Foo> pFoo;
public:
        Bar()
        {
                pFoo = std::shared_ptr<Foo>(new Foo());
        }

        std::shared_ptr<Foo> getFoo()
        {
                return pFoo;
        }
};
```

```cpp
void SomeAction()
{
        Bar* pBar = new Bar(); //with the Bar object, a new Foo is created and stored
        //reference counter = 1

        std::shared_ptr<Foo> pFoo = pBar->getFoo(); //a copy of the shared pointer is
        //reference counter = 2

        pFoo->doSomething();

        delete pBar; //with pBar the private pFoo is destroyed
        //reference counter = 1

        return; //with the return the local pFoo is destroyed automatically
        //reference counter = 0
        //internally the std::shared_ptr destroys the reference to the Foo object
}
```

```cpp
void SomeOtherAction(std::shared_ptr<Bar> pBar)
{
        std::shared_ptr<Foo> pFoo = pBar->getFoo(); //a copy of the shared pointer
        //reference counter = 2

        pFoo->doSomething();

        return; //local pFoo is destroyed
        //reference counter = 1
}
```

ENNERRE CONSULTING

Smart Pointers – weak_ptr

- weak_ptr
  - Like `std::shared_ptr`, a typical implementation of `std::weak_ptr` stores two pointers:
    - a pointer to the control block
    - the stored pointer of the shared_ptr it was constructed from
  - A separate stored pointer is necessary to ensure that converting a shared_ptr to weak_ptr and then back works correctly, even for aliased shared_ptrs
  - It is not possible to access the stored pointer in a weak_ptr without locking it into a shared_ptr

Smart Pointers – weak_ptr

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void observe()
{
    std::cout << "gw.use_count() == " << gw.use_count() << "; ";
    // we have to make a copy of shared pointer before usage:
    if (std::shared_ptr<int> spt = gw.lock())
        std::cout << "*spt == " << *spt << '\n';
    else
        std::cout << "gw is expired\n";
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        observe();
    }

    observe();
}
```

```
gw.use_count() == 1; *spt == 42
gw.use_count() == 0; gw is expired
```

ENNERRE CONSULTING

New Standard Type Library

ENNERRE
CONSULTING

C++ New Headers

Headers added in C++11				
<array>	<condition_variable>	<mutex>	<scoped_allocator>	<type_traits>
<atomic>	<forward_list>	<random>	<system_error>	<typeindex>
<chrono>	<future>	<ratio>	<thread>	<unordered_map>
<codecvt>	<initializer_list>	<regex>	<tuple>	<unordered_set>
Headers added in C++14				
<shared_mutex>				
Headers added in C++17				
<any>	<filesystem>	<optional>	<string_view>	<variant>
<execution>	<memory_resource>			
Headers added in C++20				
<barrier>	<concepts>	<latch>	<semaphore>	<stop_token>
<bit>	<coroutine>	<numbers>	<source_location>	<syncstream>
<charconv>	<format>	<ranges>		<version>
<compare>				
Headers added in C++23				
<expected>	<flat_set>	<mdspan>	<spanstream>	<stdfloat>
<flat_map>	<generator>	<print>	<stacktrace>	

<array>

- Main classes:
  - `array`
    - `operator[]`, `at()`, `front()`, `back()`, `data()`
  - `tuple_size` and `std::tuple_size<std::array>`
  - `tuple_element` and `std::tuple_element<std::array>`
  - `std::swap<std::array>`
  - `to_array`
  - `get(std::array)`
- Range access:
  - `begin`, `cbegin`, `end`, `cend`, `rbegin`, `crbegin`, `rend`, `crend`, `size`, `ssize`, `empty`

<array>

- `std::array` is a container that encapsulates fixed size arrays
  - an aggregate type with the same semantics as a struct holding a C-style array `T[N]` as its only non-static data member
    - Unlike a C-style array, it doesn't decay to `T*` automatically
    - As an aggregate type, it can be initialized with aggregate-initialization given at most `N` initializers that are convertible to `T`: `std::array<int, 3> a = {1, 2, 3};`
  - The struct combines the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc.
  - `std::array` satisfies the requirements of *Container* and *ReversibleContainer* except that default-constructed array is not empty and that the complexity of swapping is linear, satisfies the requirements of *ContiguousContainer*, and partially satisfies the requirements of *SequenceContainer*
  - There is a special case for a zero-length array (`N == 0`)
    - In that case, `array.begin() == array.end()`, which is some unique value
    - The effect of calling `front()` or `back()` on a zero-sized array is undefined
  - An array can also be used as a tuple of `N` elements of the same type

<type_traits>

- This header is part of the metaprogramming library
  - C++ provides metaprogramming facilities, such as type traits, compile-time rational arithmetic, and compile-time integer sequences
  - Type traits
    - Type traits define compile-time template-based interfaces to query the properties of types
      - Unary type traits
      - Composite type categories
      - Type properties
      - Supported operations
      - Property queries
      - Type relationships
        - Type relationship traits can be used to query relationships between types at compile time
      - Type transformations
        - Type transformation traits transform one type to another following some predefined rules

<chrono>

- Date and time utilities
  - C++ includes support for two types of time manipulation:
    - The chrono library, a flexible collection of types that track time with varying degrees of precision (e.g. std::chrono::time_point)
    - C-style date and time library (e.g. std::time)
- std::chrono library
  - The chrono library defines three main types as well as utility functions and common typedefs:
    - clocks
    - time points
    - durations

<chrono> - Clocks

- A clock consists of a starting point (or epoch) and a tick rate
  - For example, a clock may have an epoch of January 1, 1970 and tick every second
  - The Clock requirements describe a bundle consisting of a `std::chrono::duration`, a `std::chrono::time_point`, and a function `now()` to get the current `time_point`
    - The origin of the clock's time_point is referred to as the clock's epoch.
  - C++ defines several clock types:

system_clock	wall clock time from the system-wide realtime clock
steady_clock	monotonic clock that will never be adjusted
high_resolution_clock	the clock with the shortest tick period available
is_clockis_clock_v	determines if a type is a Clock
utc_clock	Clock for Coordinated Universal Time (UTC)
tai_clock	Clock for International Atomic Time (TAI)
gps_clock	Clock for GPS time
file_clock	Clock used for file time
local_t	pseudo-clock representing local time

<chrono> - Time Point

- A time point is a duration of time that has passed since the epoch of a specific clock
  - `time_point`
    - a point in time
  - `clock_time_conversion`
    - traits class defining how to convert time points of one clock to another
  - `clock_cast`
    - convert time points of one clock to another

<chrono> - Calendar

last_spec	tag class indicating the last day or weekday in a month
day	represents a day of a month
month	represents a month of a year
year	represents a year in the Gregorian calendar
weekday	represents a day of the week in the Gregorian calendar
weekday_indexed	represents the n^{th} weekday of a month
weekday_last	represents the last weekday of a month
month_day	represents a specific day of a specific month
month_day_last	represents the last day of a specific month
month_weekday	represents the n^{th} weekday of a specific month
month_weekday_last	represents the last weekday of a specific month
year_month	represents a specific month of a specific year
year_month_day	represents a specific year, month, and day
year_month_day_last	represents the last day of a specific year and month
year_month_weekday	represents the n^{th} weekday of a specific year and month
year_month_weekday_last	represents the last weekday of a specific year and month
operator/	conventional syntax for Gregorian calendar date creation

<chrono> - Others

- Duration
  - duration
    - A duration consists of a span of time, defined as some number of ticks of some time unit. For example, "42 seconds" could be represented by a duration consisting of 42 ticks of a 1-second time unit
- Time of day
  - hh_mm_ss
    - splits a duration representing time elapsed since midnight into hours, minutes, seconds, and fractional seconds, as applicable. It is primarily a formatting tool
  - Is_am, is_pm, make12, make24
- Time zone
- Literals

Multithreading and Concurrency

ENNERRE
CONSULTING

Concurrency Support Library

- C++ includes built-in support for
  - Threads
  - Atomic operations
  - Mutual exclusion
  - Futures

ENNERRE CONSULTING

Threads

- Threads enable programs to execute across several processor cores
  - thread
    - manages a separate thread
  - jthread
    - std::thread with support for auto-joining and cancellation
- Functions managing the current thread
  - yield
    - suggests that the implementation reschedule execution of threads
  - get_id
    - returns the thread id of the current thread
  - sleep_for
    - stops the execution of the current thread for a specified time duration
  - sleep_until
    - stops the execution of the current thread until a specified time point

ENNERRE CONSULTING

thread Class

- The class thread represents a single thread of execution
  - Threads allow multiple functions to execute concurrently
  - Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument
  - The return value of the top-level function is ignored and if it terminates by throwing an exception, std::terminate is called
  - The top-level function may communicate its return value or an exception to the caller via std::promise or by modifying shared variables (which may require synchronization, see std::mutex and std::atomic)
- std::thread objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may not be associated with any thread objects (after detach)
- No two std::thread objects may represent the same thread of execution
- std::thread is not CopyConstructible or CopyAssignable, although it is MoveConstructible and MoveAssignable

jthread Class

- The class jthread represents a single thread of execution
  - It has the same general behavior as std::thread, except that jthread automatically rejoins on destruction, and can be cancelled/stopped in certain situations
- Unlike std::thread, the jthread logically holds an internal private member of type std::stop_source, which maintains a shared stop-state
- The jthread constructor accepts a function that takes a std::stop_token as its first argument, which will be passed in by the jthread from its internal std::stop_source
  - This allows the function to check if stop has been requested during its execution, and return if it has

Atomic Operations

- Components provided for fine-grained atomic operations allowing for lockless concurrent programming
- Each atomic operation is indivisible with regards to any other atomic operation that involves the same object
- Atomic objects are free of data races
- Atomic types
  - atomic
    - atomic class template and specializations for bool, integral, floating-point,(since C++20) and pointer types
  - atomic_ref
    - provides atomic operations on non-atomic objects

Mutual Exclusion

- Mutual exclusion algorithms prevent multiple threads from simultaneously accessing shared resources
  - This prevents data races and provides support for synchronization between threads

mutex	provides basic mutual exclusion facility
timed_mutex	provides mutual exclusion facility which implements locking with a timeout
recursive_mutex	provides mutual exclusion facility which can be locked recursively by the same thread
recursive_timed_mutex	provides mutual exclusion facility which can be locked recursively by the same thread and implements locking with a timeout
shared_mutex	provides shared mutual exclusion facility
shared_timed_mutex	provides shared mutual exclusion facility and implements locking with a timeout

Futures

- The standard library provides facilities to obtain values that are returned and to catch exceptions that are thrown by asynchronous tasks (i.e. functions launched in separate threads)
  - These values are communicated in a shared state, in which the asynchronous task may write its return value or store an exception, and which may be examined, waited for, and otherwise manipulated by other threads that hold instances of std::future or std::shared_future that reference that shared state

promise	stores a value for asynchronous retrieval
packaged_task	packages a function to store its return value for asynchronous retrieval
future	waits for a value that is set asynchronously
shared_future	waits for a value (possibly referenced by other futures) that is set asynchronously
async	runs a function asynchronously (potentially in a new thread) and returns a std::future that will hold the result
launch	specifies the launch policy for std::async
future_status	specifies the results of timed waits performed on std::future and std::shared_future

ENNERRE CONSULTING