Classes

A class is a *type* – two syntactic choices:

- **struct** – typically for objects with public members only (“behaviourless aggregates”)
- **class** – otherwise
- *class type* is a common notation in C++ for **class**, **struct**, and **union** types

The class has *module properties*.

- encapsulates its members
- access to members can be controlled by *access specifiers* and *friend declarations*
  - **public** – access by anyone – default for **struct**
  - **private** – access by the class’ member functions only – default for **class**
  - **protected** – access by subclasses – “public” for subclasses, “private” for others
  - **friend** – can be a global function; a specific member function of another class, or a class (i.e. all member functions of that class)
  - a friend declaration is not transitive, i.e. friendship is not inherited
  - beware – friendship creates a stronger coupling than derivation

Special rules describe the scope of names declared in classes – *class scope*

- not only the declarative region made up of the class definition itself, but also, e.g. separately defined member function bodies

Class objects in C++ can be either *statically declared* objects, or *allocated dynamically* and referred by pointers.

- makes the C++ object model quite different from many other object-oriented languages

Class members

Class members can be of four kinds:

- *data members*
- *member functions*
- *nested types* – e.g. a nested **class**, a nested *alias declaration* (**using**), or a nested **typedef**
- *enumerators* – acts as static, constant data members – **enum** { Up, Down, Left, Right };

A data member or member function can be

- *non-static* – “instance member” – each object have its own copy
- **`static`** – “class member”

A member function can be

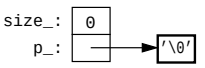
- *non-const* – allowed to modify the state of objects
- **`const`** – can not modify the state of objects
  - important to declare member functions **`const`** if they are not to modify object state
  - some **`const`** functions may need to alter a data member – declare the data member **`mutable`**
  - be “const correct”, sooner or later you will otherwise get into trouble…

Class String definition, selected parts (1)

```
class String
{
public:
   using size_type = std::size_t;              // nested type (alias declaration)

   String() = default;                         // default constructor, compiler generated
   String(const String&);                      // copy constructor
   String(String&&) noexcept;                  // move constructor
   String(const char*);                        // type converting constructor, from C string
   String(size_type, char);
   String(std::initializer_list<char>);

   ~String();                                  // destructor

   String& operator=(const String&) &;         // copy assignment operator, ref-qualifier (lvalues only)
   String& operator=(String&&) & noexcept;     // move assignment operator
   String& operator=(const char*) &;           // type converting assignment operator

   size_type length() const;                   // const member function – accessor function
   bool      empty() const;
   void      clear();                          // non-const member function – mutator function

   const char* c_str() const;                  // type conversion to C string
   explicit operator const char*() const;

   void swap(String&) noexcept;                // swap content with other String
```


Class String definition, selected parts (2)

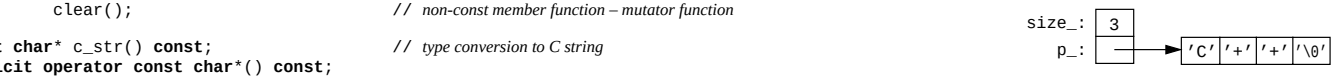
```
private:
   static char empty_rep_[1];                  // static, null-terminated C string to represent empty string

   size_type size_{ 0 };            // NSDMI
   char*     p_{ empty_rep_ };

   // Internal helper functions
   void construct_(const char*, size_type);
   void construct_(size_type, char);
   void construct_(std::initializer_list<char>);
   void append_(const char*);
   void append_(char);
};
```



`empty_rep_` is defined in the implementation file:

```
char String::empty_rep_[1];    // initialized to '\0'
```

A non-empty String has its own, dynamically allocated memory – a null terminated character array (“C string”).



Creating and operating on class objects

Class objects – variables and constants – can be either automatically och dynamically created/destroyed.

```
int main()
{
   String s;                          // variable String

   const String cs{ s };              // constant String – only const member functions can be applied

   cout << cs.c_str() << " has length " << cs.length() << '\n';     // the dot operator

   String* ps{ new String };          // dynamically created String object, pointed to by ps

   const char* p{ ps->c_str() };      // the arrow operator to access member in pointed-to object

   String& rs{ s };                   // reference to String – alternative name for s

   cout << rs << endl;                // a reference is automatically dereferenced when used
}
```

- another implicitly defined operator is the important scope resolution operator ::

```
class::member              // qualified name
```

- there are quite a few more implicitly defined operators

Definition of member functions

- separately, outside the class definition, e.g on an accompanying implementation file `String.cc`

```
#include "String.h"
…
String::size_type  String::length() const
{
   return size_;
}
```

  - `String::` must precede the name of any separately defined member – a *qualified name*
  - also when from outside class scope referring to a member, such as `size_type`
  - the parameter list and the function body is within class scope of String
- within the class definition (*inclass definition*)

```
class String
{
   …
   size_type length() const { return size_; }
   …
};
```

- member functions defined *inclass* are automatically *inline* functions
- separately defined member functions can explicitly be declared **inline**

Special member functions

- *default constructor* – initializes a new object without any arguments
- *copy constructor* – initializes a new object from an existing object of the same type
- *move constructor* – typically used if the source is a temporary object of the same type
- *copy assignment* – assigns from an object of the same type
- *move assignment* – typically used if the right hand side is a temporary object of the same type
- *destructor* – cleans up when an object cease to exist, typically releasing resources

Compiler generated (*implicitly declared/defined*) if not declared by programmer (there are detailed rules), with the following semantics:

- *default constructor*
  - data members with default initialization (class type members) are default initialized, in declaration order
  - data members without default initialization are not initialized, e.g. fundamental type members (**int**, **double**, **pointers**, etc.)
- *copy constructor*,
  - member-by-member copy construction from source to destination, in declaration order
- *copy assignment*
  - member-by-member assignment from source to destination, in declaration order
- *move constructor*, *move assignment*
  - instead of copying content, it is *moved* from the source to the destination
- *destructor*
  - data members with destruction (class type members) are destroyed, in reverse declaration order

Constructors

- one is always invoked automatically when class objects are created
- shall initialize the objects to a well-defined state
- cannot be declared to return anything, not even **void**
- can have parameters – these can have default arguments – can be overloaded – a class can have several constructors

```
String(const char*);
String(const String&);
String(String&&) noexcept;
String(std::initializer_list<char>);
```

- *default constructor* – a constructor which can be *used* without any arguments

```
String(const char* s = "");          // if no explicit argument, "" will be used  – "two-in-one"
```

- *copy constructor* – a constructor with a (first) parameter of the class type – to create a new object as a copy of another

```
String(const String& other);         // creates the new object as a copy of other
```

- *move constructor* – a constructor with a (first) parameter of type rvalue reference – to create a new object by emptying another

```
String(String&& other) noexcept;     // creates the new object by pilfering the content of other
```

- *type converting constructor* – a constructor which can be invoked with one argument of another type

```
String(const char* s);               // converts C string to String
```

Using constructors

Selected by overload resolution.

```
{
   String s1;                              // default constructor called – empty String created

   String s2{ "C++" };                     // initializing with string literal – type conversion

   String s3{ s2 };                        // initializing with another String object – copy

   String* p{ new String{"C++"} };         // constructor taking a C string called

   vector<String> v(10);                   // 10 container elements – default constructor used for each

   String a[10];                           // array element – default constructor is used for each element

   String s5{ 'C', '+', '+', '1', '1' };   // constructor taking initializer list
}
```

Note – the following declaration does not create a default constructed String `s`

```
String s();
```

but this expression does, a temporary object

```
return String();
```


Member initializers

A *member initializer list* can be used in constructors to initialize data members before the constructor body is executed.

Alternative implementation for String(**const char*** str), possible if we could be sure the parameter str in never **nullptr**:

```
String(const char* s)
   : size_{ strlen(s) }, p_{ strcpy(new char[size_ + 1], s) }
{}
```

- if both an initilizer in the data member declaration, and a member initializer as above, only the member initializer will be executed
- a member initializer list is inserted between the constructor parameter list and the constructor body
  - starts with a colon
  - initializers are separated with comma, if more than one
- the data members are initialized in *declaration order*, not in the order initializers are written
  - `size_` must be declared before `p_` , since the value of `size_` is used in the initializer for `p_`
  - always write member initializers in the same order as the members are declared
- a member with default initialization will always be default initialized before the execution enters the constructor body
  - use initializer, if possible, instead of assignment in constructor body, to avoid “double” initialization
- **const** and reference members cannot be assigned in the constructor body
  - follows from ordinary semantic rules – such types must always be initialized when they are created
- member initializer is the only way to invoke a base class constructor from a subclass constructors

```
Derived(parameters) : Base(arguments), … {}
```


Destructor

- called automatically when an object is on its way to disappear
  - for a dynamically created object (**new**) when deleted (**delete**)
- typically used to release resources, e.g. deallocate memory, or closing a file

```
~String() { if (!empty()) delete[] p_; }
```

- cannot be declared to return anything – not even **void**
- have no name – special declaration syntax
- cannot have parameters – cannot be overloaded – a class can only have one destructor
- explicitly called only in special cases

```
{
   String s;                          // s created automatically – constructor invoked

   String* p{ new String };           // Object created dynamically – constructor invoked
   …
   delete p;                          // Dynamic object deallocated – destructor implicitly invoked
   p = nullptr;
   …
}  // declaration block for s is terminated – s disappears – the destructor is implicitly invoked
```


The this pointer

One problem with the calling syntax for member functions is that the object don’t belong to the parameter list

```
s.at(7);
```

as in a traditional function call

```
at(s, 7);
```

where the object s is accessible in the function through the corresponding parameter.

- what if you need to refer to the object in question in a member function – use the **this** pointer
  - in a non-static member function **this** points to the object for which the function is called
  - in a non-**const** member function **this** have type `String*`
  - in a **const** member function **this** have type `const String*`
  - **volatile** can also appear, e.g. `const volatile String*`
  - *move semantics* can also be applied to ***this** (comes later)
- **this** is usually used implicitly to access members – an explicit reference is written, e.g.

```
size_type String::length() const
{
   return this->size_;
}
```

  - length() is **const** – the **this** pointer have type **const** String* – the object is **const** in this context – size_ cannot be modified

Default constructor

Since there are other constructors declared, a default constructor would not be generated, according to rule.

- a naïve, straight-forward way to define the default constructor would be

```
String() {}
```

  - the data members will be initialized according to the initializers in their declarations
- by *defaulting*, we ask the compiler to generate

```
String() = default;
```

  - a defaulted function can be more efficient than a hand-written one

Type converting constructor

A constructor taking *one* argument of another type defines a type conversion, possibly implicit.

- declaring **explicit** means that it cannot be invoked to make implicit type conversions

```
class String
{
public:
   String(const char*);       // not explicit
   …
};
```

- If **operator**=(**const char***) had *not* been declared for String, the assignment below would still be allowed

```
s = "This is a literal of type const char*";
```

  - semantically equivalent to

```
s = String("This is a literal of type const char*");
```

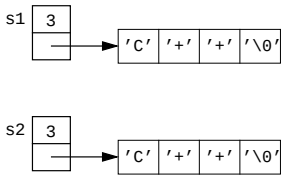
  - the literal of type **const char*** is converted to a temporary object of type String this constructor
  - **operator**=(String&&) can then be applied to `s` and the temporary object

Copy constructor

Deep copy.

- the compiler generated constructor would just copy size_ and p_ – the character array would be shared by several object
- a new object is to be created – no history to take into account if something goes wrong
- the helper function construct_ takes care of the greedy details – memory allocation and copying value from other

```
String::String(const String& other)
{
   construct_(other.p_, other.size_);
}

void String::construct_(const char* cstr, size_type size)
{
   if (cstr != nullptr && size > 0)
   {
      p_ = strcpy(new char[size + 1], cstr);
      size_ = size;
   }
}

String s2{ s1 };         // direct initialization syntax

String s2 = s1;          // copy initialization syntax
```

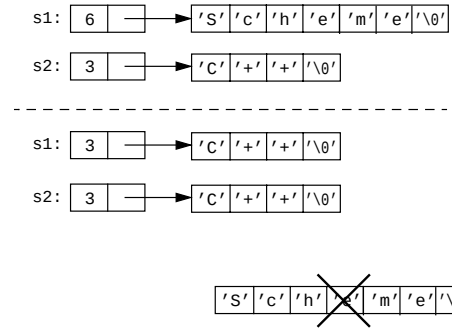


  - allocating memory for the character array is the critical part – if **new** throws (bad_alloc), just bail out – no cleanup required
  - copying size is safe – memory for the String object itself (size_ and p_) is already there, assigning **int** will not fail

Copy assignment operator

Deep assignment.

```
s1 = s2;
```



- left hand side has a history
  - dispose of old content
  - copy new value from right hand side
- important to do things in the right order
  - no object should end up in an undefined state
  - make sure to get new memory first
  - then make changes

Copy assignment operator – straightforward implementation

- the default copy assignment would just assign size_ and p_ – the character array from source object would be shared
- the destination is an existing object – old content to take care of – otherwise the same copy semantics as for the copy constructor

```
String& String::operator=(const String& rhs) &          // ref qualifier – assign to lvalue only
{
   if (this != &rhs)
   {
      char* p{ empty_rep_ };                            // in case rhs is an empty String
      if (!rhs.empty())
         p = strcpy(new char[rhs.size_ + 1], rhs.p_);    // if we survive this we're fine
      size_ = rhs.size_;
      if (!empty()) delete[] p_;
      p_ = p;
   }
   return *this;
}
```

- checks if the left and right hand side operands are the same object – self-test

```
s = s
```

- performs a deep copy in a strongly exception-safe way – allocates memory before anything is changed – if **new** throws
  - no memory will leak
  - none of the objects will be corrupted
- semantics corresponds to built-in assignment – returns a non-const reference (*lvalue*) to the left-hand side argument

Copy assignment operator – elegant version

- uses the idiom "create a temporary and swap"
- need a function that can exchange the contents of two objects in a safe way, preferably a *nothrow swap*

```
String& String::operator=(const String& rhs) &
{
   String{ rhs }.swap(*this);     // create a temporary and swap
   return *this;
}
```

- a temporary is created and initialized by the copy construct – deep copy of rhs
- the contents of **this** and the temporary is swapped
  - **this** now becomes a copy of `rhs`
  - the temporary takes over of the old content of **this** – especially the character array
- the temporary is destroyed after the swap
  - the old dynamic memory for **this** is deallocated
- if an exception is thrown, it will happen when the temporary is initialized
  - strongly exception-safe – no memory will leak – no object will end up in an undefined state
  - exception neutral – any exception thrown is propagated further as is

Note. If the elements had been of class type, exceptions could also be thrown when copying the elements,

Move semantics

One of the big news in C++11 – alternative to classic *copy semantics*.

- *temporary objects* are created in different situations – often implicitly
- if such an object is used to initialize or assign another object it is unnecessary to make a copy
  - copying can be costly – time and space
  - instead move the content of the temporary to the destination object
- how find such objects – they are not visible?
  - the compiler knows!
  - *rvalue-references* catches them automatically!
  - we just need to be aware of the possibility and have it in mind when we construct classes

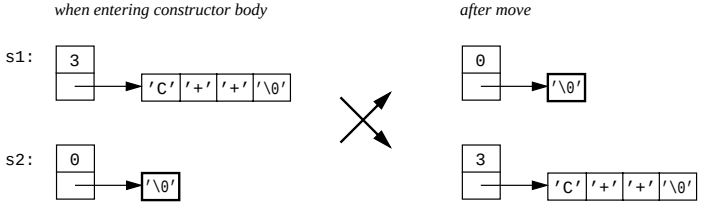
```
String(const String&);                  // this can catch all kind of objects but
String(String&&) noexcept;              // this one is a better match for temporary objects (rvalues)

String& operator=(const String&) &;
String& operator=(String&&) & noexcept;
```

- implementing move semantics
  - an object which resources have been moved must be *destructible*
  - sometimes we want to apply move semantics also to ordinary objects (*lvalues*)
  - an object which have been moved from must be *assignable* (and *copyable*)
  - the principle should be that an object moved from should be comparable to a default initialized object

Move constructor

Our implementation use the member functions swap():

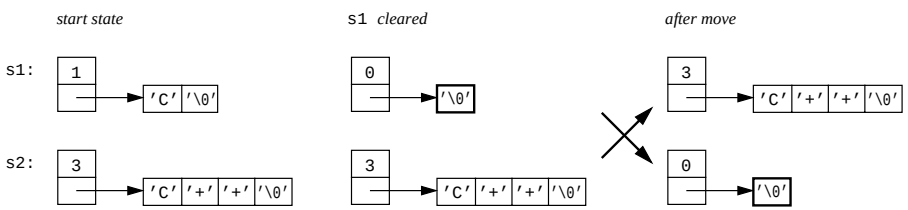
```
String::String(String&& other) noexcept     // size_ and p_ are initialized by their NSDMI – empty string
{
   swap(other);                             // swap content with other
}

String s1 = String{ "C++" };                // temporary object (rvalue)

String s2{ std::move(s1) };                 // utility function move() converts s1 (lvalue) to String&& (rvalue)
```



Move assignment operator

Our implementation uses clear() and swap():

```
String& String::operator=(String&& rhs) & noexcept
{
   clear();              // the left hand side operand is cleared – set to ”empty string”
   swap(rhs);            // move by swapping content with the right hand side operand
   return *this;
}

s1 = std::move(s2);
```



An alternative is to just swap contents for the two objects (there is no precise definition of move semantics).

Rules for move constructor and move assignment operator generation

- The *move constructor* is only generated if, the class does
  - *not* have a user declared copy constructor
  - *not* have a user declared copy assignment operator
  - *not* have a user declared move assignment operator
  - *not* have a user declared destructor
- The *move assignment operator* is only generated if, the class does
  - *not* have a user declared copy constructor
  - *not* have a user declared move constructor
  - *not* have a user declared copy assignment operator
  - *not* have a user declared destructor

Style recommendations for declaring special member functions

- if a special member function **is** desired, and the compiler **can** generate it – *default it*
- if a special member function is **not** desired, and the compiler **will** generate it – *delete it*
- if a special member function is **not** desired, and the compiler will **not** generate it – *don't declare it*

The name rule

Things that are declared as *rvalue reference* can be either an *rvalue* or an *lvalue*. The name rule says:

- if it has a *name* it's an *lvalue*
- *otherwise*, it's an *rvalue*

```
void foo(T&& x)          // rvalue reference having a name, x
{
   T  y{ x };            // x is an lvalue, T::T(const T&) is called
   …                     // x still in scope – it would be dangerous to allow move semantics to be allowed tacitly
}

T&& fie();               // rvalue reference having no name

T  x{ fie() };           // T::T(T&&) is called

T fum();                 // rvalue having no name (temporary)

T  y{ fum() };           // T::T(T&&) is called
```


Utility function std::move()

The traditional way to exchange values for two String variables:

```
void swap(String& x, String& y)
{
   String tmp{ x };              // x is copied to tmp by the copy constructor
   x = y;                        // y is copied to x by the copy assignment operator
   y = tmp;                      // tmp is copied to y by the copy assignment operator
}
```

Both x and y shall receive new values – their old values is not required to be kept when they are copied – *move instead*

```
#include <utility>

void swap(String& x, String& y)
{
   String tmp{ std::move(x) };   // x is moved to tmp by the move constructor
   x = std::move(y);             // y is moved to x by the move assignment operator
   y = std::move(tmp);           // tmp is moved to y by the move assignment operator
}
```

- std::move() doesn't really do more than type convert to an *rvalue reference* – `String&&` in this case
- this will make the move versions to be chosen instead of the copy versions

Note: move() basically applies `static_cast<String&&>(`*arg*`)`

Ref-qualifiers for non-static member functions

Extends move semantics to ***this** – thee options:

- no ref-qualifier
- **&** ref-qualifier, same as no ref-qualifier
- **&&** ref-qualifier

```
struct X
{
   void fun() &;           // OK, *this will be X& – lvalue reference
   void fun() &&;          // OK, *this will be X&& – rvalue reference
   void fun() const &;     // OK, *this will be const X& – lvalue reference to const
};
```

- **&** and **&&** can be overloaded, in combination with the cv-qualifers

```
struct X
{
   X* operator&() &;              // selected for lvalues only
   X& operator=(const X&) &;      // selected for lvalues only
};

X* p = &X();      // error: takes the address of something temporary (rvalue)

X x;

X() = x;          // error: no known conversion for implicit this parameter from X to X&
```


Situations where special member functions are used

```
String  g1{ "Global" };     // g1 is initialized by the type converting constructor for C string

String  g2{ g1 };           // g2 is initialized by the copy constructor – direct initialization syntax


String fun(String s)        // s is initialized by the copy constructor, unless the argument is a temporary, then move
{
   String  loc{ s };           // loc is initialized by the copy constructor

   loc = s;                    // loc is assigned by the copy assignment operator

   return loc;                 // a temporary object is created and initialized by the move constructor
}                              // the local objects s and l are destroyed by the destructor


int main()
{
   String  g3 = g1;         // g3 is initialized by the copy constructor – copy initialization syntax

   g2 = fun(g1);            // g2 is assigned from a temporary – invokes the move assignment operator
}
```

Sometimes the temporary used for passing a return value from a function can be *elided*.

- in such case the value of `loc` would be directly copied or moved into a destination object
- a common optimization called RVO – Return Value Optimization

Delegating constructors

A constructor can *delegate* to another constructor of the same class.

- if empty Strings had been represented in the same way as non-empty Strings, the default constructor could have been written:

```
String() : String{ "" } {}          // delegate to the constructor below

String(const char*);
```

- an alternative is to let the type converting constructor also represent the default constructor (as we have chosen)

```
String(const char* = "");
```


Ways to restrict and control object creation (1)

Some objects, e.g. stream objects, are not suitable to copy.

- copy construction and copy assignment can be disallowed
- declare **delete** to remove a special member functions,
- **default** if you want the compiler to generate

```
public:
   X() = default;                      // default initialization allowed

   X(const X&) = delete;               // no copy construction

   X& operator=(const X&) = delete;    // no copy assignment
```

- if, e.g., the compiler generated copy constructor is fine, but you want to restrict it for internal use only, **default** it as **protected**

```
protected:
   X(const X&) = default;
```

- if at least one of the destructor, copy constructor or copy assignment operator is declared, the move constructor or move assignment operator are *not* generated
  - the move constructor and move assignment operator should never be deleted – either declare, if desired, or ignore, if not
- eliminating public constructors does *not* make a class *abstract* (derived classes)

Note: a *defaulted* function must be a member function, a *deleted* function need *not* be.

Ways to restrict and control object creation (2)

Dynamic memory allocation can be disallowed by hiding the predefined *global allocation functions*.

```
private:
   static void* operator new(size_t) = delete;                    // plain versions
   static void  operator delete(void*, size_t) = delete;

   static void* operator new[](size_t) = delete;                  // array versions
   static void  operator delete[](void*, size_t) = delete;
```

- these are always static members, even if not explicitly declared so
- there are also *nothrow* and *placement* versions of global **new**/**new**[] and **delete**/**delete**[]

In the context of derived classes there is also the concept of *abstract class*, for which no objects can be created, except as subobjects of objects of a derived concrete class.

For a **new** *expression*, such as '**new** String', one of these **new** functions will be called, resulting in a two-step procedure:

- memory is acquired, and, if successful, then
- the constructor is executed to create the object in that memory

For a **delete** *expression*, such as '**delete** p', one of these **delete** functions will be called, also resulting in a two-step procedure:

- the destructor is executed to destroy the object in the memory
- the memory is released

Type conversion functions

Type conversion *from* a class to some other type is defined as a *conversion function*.

- can be declared **explicit** to disallow implicit use

```
class String
{
public:
   …
   operator const char*() const { return p_; }          // not explicit
   …
};

// If not present: ostream& operator<<(ostream&, const String&);

String s{ "Hello World!" };

cout << s << endl;              // Implicit type conversion to const char*
```

- this type conversion function makes it possible to use predefined **operator**<< for **const char***, if not declared **explicit**
  - such conversion can cause much more problems than it solves
  - be aware of possible type conversion sequences for arithmetic types
  - **explicit** solves such problems but allows explicit use, e.g.

```
cout << static_cast<const char*>(s) << endl;
```

- an ordinary member function doing the same can be an alternative, or a complement `– c_str()`

Static members

`String` have a static data member to represent the empty string value.

- member functions and data members declared **`static`** are common for all object belonging to a class – *class members*.

```
class C
{
   static int s;                    // static data member declaration
   static int get_s();              // static member function declaration
};

int C::s{};                         // static data member definition, explicitly initialized

int C::get_s() { return s; }        // static member function definition
```

- Static members can be accessed in two ways:
  - class name and the scope operator (*qualified name*)

```
C::get_s()
```

  - an object and the *dot operator*, or a pointer to an object and the *arrow operator*

object`.`*member*

pointer-to-object`->`*member*

A class, its operations, namespaces, and name lookup

A nonmember function belonging to a class is an operation of that class, no less than a member function.

- natural to declare such non-member functions in the same namespace as the class
- *argument dependent lookup* (ADL) is applied to unqualified function names and depend upon the arguments given in the call
  - if a member function turns up, ADL does not occur
  - the set of namespaces searched during ADL depends upon the types of the function parameters
  - at least namespaces containing the parameter types belong to the set of searched namespaces
- there are known problems – ADL may lead to unpleasant "surprises"

A namespace is a simple module construct.

- encapsulates the declarations within it – access controlled by *using directives* and *using declarations*

```
using namespace std;        // opens the entire standard namespace – all names becomes directly visible

using std::string;          // introduces the name string, as if it was declared at this point
```

- can be added to, successively
- have influence on name lookup

String and namespace

String is encapsulated in a namespace named `IDA_String`.

- first introduced in `String.h` (*original namespace definition*)

```
namespace IDA_String
{
    class String { … };
    …
}

#endif
```

- added to in `String.cc` (*extension namespace definition*)

```
#include "String.h"

namespace IDA_String
{
    Separate definitions for String member functions
}
```

- a *qualified name* including the namespace name can alternatively be used:

```
IDA_String::String::size_type  IDA_String::String::length() const { … }
```


String iterators

String is a container storing elements of type **char**, and should as such have iterators.

```
for (auto it = begin(s); s != end(s); ++i) …
```

- String iterators can be defined as character pointers and implemented as random access iterators

```
using iterator               = char*;
using const_iterator         = const char*;
using reverse_iterator       = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
```

  - iterator operators are given by the ordinary pointer operators: `*`, `->`, `++`, `--`, etc.
  - std::reverse_iterator is a standard template utility class to define reverse iterators, given the forward iterators `iterator` and `const_iterator`
  - usually `iterator` and `const_iterator` must be defined as classes
- the full set of iterator member functions are defined, some examples

```
iterator       begin() { return iterator(p_); }
const_iterator begin() const { return const_iterator(p_); }
iterator       end() { return iterator(p_ + size_); }
const_iterator end() const { return const_iterator(p_ + size_); }
```

Iterators will be covered in the standard library lectures.

Some recommendations for single class design (1)

- a class should do one thing, and do it well
- members not to be accessed directly should be **private** (or **protected**)
- member functions that does not modifying the state of the objects shall be **const**
- can the constructors initialize the object in all desired ways?
- declare and initialize data members in the same order.
  - the declaration order determines the initialization order, so be consistent with that
- prefer initialization to assignment in constructors
  - often more readable
  - avoids “double initialization” of default initialized members
- does the compiler generated copy constructor, copy assignment operator, and destructor work, or must they be defined?
- copy and destroy consistently
  - if you write/disable the copy constructor or the copy assignment operator, you probably need to do the same for the other
  - if you write copy functions, you probably need to write a destructor, and vice-versa
- explicitly enable or disable copying (copy constructor and copy assignment)
  - if desired and compiler versions works fine – **default** both
  - if not desired – **delete** both
  - if desired but the compiler generated versions does not work correctly – write both
  - deleting a function declares it **private** – don’t **delete** move constructor or move assignment operator if their copy versions are allowed

Some recommendations for single class design (2)

- make data members private, except in behaviourless aggregates (C-style structs)
  - `std::pair` is defined as a **struct** with only public members
- don’t give away your internals
  - let e.g. member access functions return *reference to const* or *pointer to const*
- avoid providing implicit conversions
  - eliminates or minimizes creation of temporary objects by type converting constructors and function
  - prefer to declare converting constructors and conversion functions **explicit**
  - define optimized operator overloadings for the mixed type operation to be allowed
- whenever it makes sense, provide a no-fail swap
  - nice things can be done using swaps
- design and write error-safe code
- don’t optimize prematurely – don’t inline by default, it leads to higher coupling
  - profilers are *good* at telling you which functions you should mark **inline**
  - profilers are *bad* at telling you which functions you should *not* have marked **inline**
  - **inline** may not even matter for your compiler…