Classes and Namespaces

- Abstract Data Types
- C++ Classes
  - Constructors
  - Destructors
  - Member Functions
  - Friend Functions
  - Operator Overloading
- Namespaces

Abstract Data Types (ADT)

- Defines data organization and data handling operations
- *Data abstraction* used to define domain and structure of data along with a collection of operations that access the data
- Bundle data with operations that manipulate the data

Abstract Data Types (ADT)

- Design for each operation
  - preconditions
  - input values provided
  - process performed
  - postconditions
  - output values returned
- Implementation independent
  - reusable
  - tested separately

OOP (Object Oriented Programming)

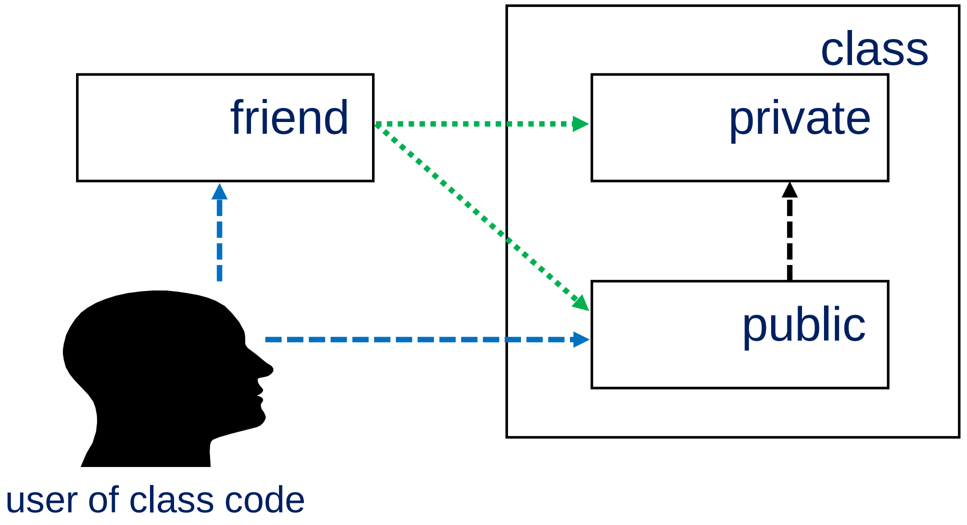
- **Encapsulation** bundles data items and methods into single entity
  - information hiding used to restrict outside access & protect data integrity
    - data protected from accidental corruption
    - code outside object does not need to know about internal structure
      - internal code changes do not need to affect external code
- **API** (application programming interface)
  - function prototypes for constructors and other public member functions of ADT

ADTs and C++

- **C++ Class** used to represent ADT and variables of class type are called **objects** (or instances)
- Class consists of **members** that include data values and operations for handling data
  - Data values can be other objects
    - **Composition classes** have access to member functions in component objects (*code reuse*)
  - Operations are also called **methods**
  - Data values visible to all methods *(class scope)*

API and C++ Classes

- Public members accessible by *all* code outside class scope
- Private members *only* accessible within class scope
  - Default
- Declared friends can access (*public and*) private members



C++ Classes

- Class is a user-defined data type that consists of **members**
  - Data items
  - Functions (**methods**)
  - By default all members are private and only have class scope
- Function definitions can be
  - Within class body (inline code)
  - Outside class body
    - Need **scope resolution operator**
- Function definitions with **const** qualifier do not modify any data items in the class
  - *Parameters* with **const** qualifier do not modify *parameter*

C++ Classes

- Class **declaration** declares class structure without defining member functions
  - Header file (*.h)
  - Inline functions cause code expansion
- Class **implementation** defines methods
  - Implementation file (*.cpp)
  - Each function must include the class scope operator::, which designates class membership

Class Constructors

- **Constructor** functions
  - Same name as class
  - Implicitly called when object created
    - Called for each element in an array
  - Do not return a value
  - Used to assign initial data values
    - **Member initialization list**
      - Comma-separated list of class member data names followed by initial value enclosed in parenthesis
      - Placed after function header and separated from parameter list by colon

Class Constructors

- **Constructor** functions
  - Can have multiple constructors
    - Differ by number and type of parameters *(overloading)*
  - **Default** constructor has *no* parameters
    - Or all parameters have defaults
  - **Copy** constructor takes object of *same type* as parameter

Copy Constructor

- Default copy constructor automatically created if not defined by user
  - Simple memberwise assignment
- Parameter type must be call by reference
  - Infinite chain of copy constructors if not!
- For classes with *dynamic* data members
- Called when
  - Object declared and assigned another object

```
        class-name obj2(obj1), obj3 = obj1;
```

  - Function parameter is pass by value
  - Function return value

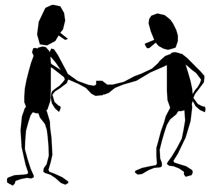
Class Destructor

- **Destructor** function
  - Often used to delete dynamic memory or decrement object counters
  - Has same name as class, prepended with a ~ (*complement*)
  - Implicitly called when object goes out of scope or memory released
  - *Cannot* pass parameters to destructor function
  - *Does not* return a value

Member Functions

- Format similar to function prototype in class declaration
  - Inline member functions include definition
    - Causes code substitution
- Include class name and scope resolution operator with definition
- Access to *all* object data
- Private data typically has
  - Public mutator function (aka setters)
  - Public accessor functions (aka getters)

Friend Functions



- Not a class member function, but has *direct* access to *all* object data
- Precede function prototype in class declaration with keyword ‘friend’
- Define function outside of class
  - Use object variable name and dot operator to access private members

Operator Overloading

- Standard operators can be redefined to work with objects:

```
++box1
box1 < box2
```

unary operator
binary operator

- Conditions of operator overloading
  - must obey rules of precedence, associativity, and number of operands
  - cannot create new operator symbols
  - cannot have default arguments

Operator Overloading

- Operators that can be overloaded

+	-	*	/	%	++	--
+=	-=	*=	/=	%=		
^	&	|	~			
^=	&=	|=				
=	<	<=	>	>=	==	!=
&&	||	!	<<	>>	<<=	>>=
,	->	->	*	[]	()	
new	delete					

- Operators that cannot be overloaded

?:	.	.*	::

Operator Overloading

- Overloaded operator must have argument of class/structure type
  - pass by reference with const qualifier *usually* for efficiency
    - unless needed for operator type
      - compound assignment
      - remove **const** keyword
- Ways to overload operators
  - free function
  - friend function
  - member function

Operator Overloading

uses public accessor and mutator methods for private member access

- As **free** function
  - unary

```
return-type operator op (const
  class& obj);
```

  - binary

```
return-type operator op (const
  class& lhs, const class& rhs);
```

Operator Overloading

- As **friend** function
  - unary

```
friend return-type operator op (const
  class& obj);
```

  - binary

```
friend return-type operator op (const
  class& lhs, const class& rhs);
```

access private members
directly with dot operator

Operator Overloading

- As **member** function
  - unary

    return-type **operator** *op* ();
  - binary

    return-type **operator** *op* (**const** *class***&** *right*);
- If overloading operator as **member** function
  - object is an operand
    - unary operator has 0 arguments
    - binary operator has 1 argument
      - object is left operand

access rhs private members directly with dot operator

Namespaces

- Types of identifier scope:
  - global
  - namespace
  - class
  - local
- Namespaces are an ANSI C++ feature that allows grouping of entities into a narrow scop

  ```
  using namespace std;
  ```
- Reduces name clashes
  - many towns have a ‘Main’ street
  - could have global instructor name in CIS2541 namespace and another global instructor name in CIS2542 namespace

Namespaces

- Any items not placed in a namespace become part of global namespace
- Syntax for declaration:
  - Namespace keyword followed by identifier *(the namespace name)*, followed by entity declarations enclosed in braces

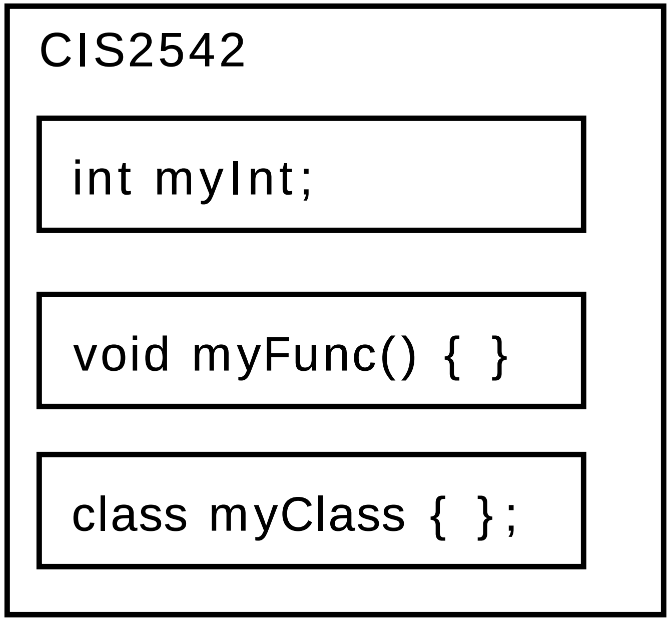
```
namespace namespace_name
{
  declarations
}
```

- Namespace declarations *can be split* across files
  - Declarations additive across files

Namespace Example

- Namespace definition

```
namespace CIS2542
{
  int myInt;
  void myFunc() { . . }
  class myClass { . .};
}
```



Referring to a Namespace

- Using full member name, including the namespace it belongs to:

```
CIS2542::myInt;
CIS2542::myFunc();
CIS2542::myClass c1;
```

- Taking advantage of Using-Declarations:

```
using CIS2542::myInt;
using CIS2542::myFunc;
using CIS2542::myClass;
myInt;
myFunc();
myClass c1;
```

Referring to a Namespace

- Taking advantage of Using-Directives:

```
using namespace CIS2542;
myInt;
myFunc();
myClass c1;
```

- Using aliases:

```
namespace alias
namespace C = CIS2542;
C::myClass c1;
or individual member alias
namespace CMC = CIS2542::myInt;
CMC c1;
```

std namespace

- According to the ANSI C++ standard, definition of all classes, objects, and functions of the standard C++ library (like **`cout`**, **`string`**, etc.) are defined within namespace **std**
- Traditional header files (iostream.h) have new specified names (iostream)
- **Highly** recommended for users of the Standard Template Library (STL)