

Week6 : Functions Part II and Classes

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 6

Outline

Functions Part II

classes

Outline

Functions Part II

- Functions with varying parameters

- Return types

- Overloaded functions

- Function defaults, inline and `constexpr`

- Pointers to functions

classes

Initializer list parameters in C++ 11

- ▶ If all argument types take the same type of object we can use `initializer_list` library type
- ▶ An `initializer_list` represents an array of specified type
- ▶ Items in `initializer_list` are always `const`
- ▶ Example use as list of strings to print out:

```
1 void print_strings(initializer_list<string> strs){
2     for ( const auto iter = strs.begin();
3         iter != strs.end; ++iter ){
4         cout << *iter << " " ;
5     }
6     cout << endl;
7 }
8 int main(){
9     initializer_list<string> sl{ "This", "is", "a",
10    "list", "of", "strings", "to print" };
11    print_strings( sl );
12    return 0;
13 }
```

Ellipsis Parameters

- ▶ Should only be used for types common to C and C++
- ▶ Ellipsis parameter may only appear as last element in a parameter list
- ▶ Commonly used in C for formatted scanning of input `scanf` and printing `printf`:

```
int scanf ( const char * format, ... );  
int printf(const char* format, ...);
```
- ▶ Example, `printf` takes a format c-array string that tells the function what values to look for in the variable number of additional parameters, eg:

```
1 #include <stdio>  
2 int main() {  
3     int i = 5;  
4     double pi = std::acos(-1);  
5     char myname[] = "Dr. B. Jamieson";  
6     printf( "Name: %s Number of bikes: %d Favorite  
           number %10.5f\n", myname, i, pi );  
7     return 0;  
8 }
```

printf format specifiers

- ▶ Each format specifier is of the form:
`%[flags][width][.precision][length]specifier`
- ▶ `flags`, `width`, `precision` and `length` are optional specifications.
- ▶ `flags`: `-` to left justify, `+` to precede result with a `+` or `-`, `space` puts a space in front of positive, `0` to left-pad number with zeros.
- ▶ `width` is the minimum number of characters to print, otherwise pad with spaces
- ▶ `precision` precision is number of digits after decimal place, or minimum number of digits for integer types

	specifier			
length	d i	u o x X	f F e E g G a A	s
(none)	int	unsigned int	double	char *
hh	signed char	unsigned char		
h	short int	unsigned short int		
l	long int	unsigned long int		
ll	long long int	unsigned long long int		
L			long double	

Functions with no return value

- ▶ Functions that have no return value are specified with keyword `void` as the return type
- ▶ `void` functions use a `return` to exit the function at an intermediate point
- ▶ Example, function to swap place of two integers

```
1 void swapint ( int &i1 , int &i2 ){
2     if ( i1 == i2 ) return ;
3     int tmp = i2 ;
4     i2 = i1 ;
5     i1 = tmp ;
6     // no explicit return required
7     return ; //optional
8 }
```

Functions that return a value

- ▶ Return within function must be of type specified.
- ▶ Example: Search for index of first matching integer.

```
1 int findindex( int match, const vector<int> & vec
   ) {
2   for ( decltype( vec.size() ) i = 0; i <
       vec.size() ; i++){
3     if ( match == vec[i] ) return i;
4   }
5   // missing return statement...
6 }
```

- ▶ Failing to provide a **return** after a loop containing a **return** is an error – compiler may or may not detect this error.

How values are returned

- ▶ Returned value is done in same way as parameter initialization
- ▶ Value initializes a temporary variable at the call site
- ▶ It is wrong to return a reference or pointer to an object that is an automatic type defined in the function
- ▶ Examples:

```
1 // return reference to shorter of two strings
2 // Okay since reference is to calling object
3 const string & shorterString( const string &s1,
4     const string &s2 ){
5     return s1.size() < s2.size() ? s1 : s2;
6 }
7 // Example error returning ref. to local object
8 const string & getEmptyStr1(){
9     string ret;
10    return ret; // reference to local object!
11 }
12 // Example error returning ref. to local object
13 const string & getEmptyStr2(){
14    return "Empty"; // empty is local temporary
15    string!
```

Class types as a function return

- ▶ Call operator has associativity and precedence that is same as `.` and `->` operators, and are left associative.
- ▶ Result below is to call `shorterString` function, then returned `string` has its `size()` method called.

```
1 auto minLen = shorterString( s1, s2 ) . size();
```

- ▶ In any case, for class return types, can then call their member function.

Reference return values

- ▶ Calls to functions that return reference are lvalue
- ▶ That means you can assign a value to it, for example:

```
1 // never call this function with loc >= str.size()
  as
2 // no check of str size is made.
3 char & get_val ( string & str ,
4                 string::size_type loc ) {
5     return str[loc];
6 }
7 int main() {
8     string s = "some long enough string";
9     cout << s << endl;
10    get_val( s, 0 ) = 'S'; // set s[0] to 'S'
11    cout << s << endl;
12    return 0;
13 }
```

- ▶ Note that function returned a non-const, so setting value was possible
- ▶ Can't set value if return value is const:

```
1 // error below trying to set const value
2 shorterString( "hi", "bye" ) = "X";
```

List initializing return value (C++ 11)

- ▶ Can return a list initialized return value in C++ 11
- ▶ Examples:

```
1 vector<string> getStr1 () {  
2     return { "String", "one", "is", "list", "init" };  
3 }  
4 vector<string> getStr2 () {  
5     string str1 = "String";  
6     string two = "two";  
7     return { str1, two, "also", "list", "init" };  
8 }
```

- ▶ If returning list init for built-in type, result must not narrow

return from main

- ▶ `main` function may come to end with no return value; it then implicitly inserts a zero return value
- ▶ Also have return values specified in `cstdlib` header file:

```
1 #include <cstdlib>
2 int main( ){
3     if ( some-failure ){
4         return EXIT_FAILURE;
5     } else {
6         return EXIT_SUCCESS;
7     }
8 }
```

Returning a pointer to an array

- ▶ Can't return whole array, so return pointer or reference to array
- ▶ Syntax is a bit ugly, so can use `typedef` to make it look nicer

```
1 // below: arrT is synonym for array of 10 integers
2 typedef int arrT[10];
3 // below: same as above
4 using arrT = int[10];
5 // below: function returning pointer to array of
6 //       10 int
7 arrT * func( int i );
```

- ▶ without an alias it is similar to pointer to array declaration:

```
1 int *p1[10]; // p1 is array of 10 ptrs
2 int (*p2)[10]; // p2 points to array of 10 ints
3 // below: func returns ptr to array of 10 ints
4 int ( *func( par , list ) )[10] ;
```

Using trailing return type

- ▶ In C++ 11 can use `auto` in place of return type, and put actual return type after function with `->` operator.
- ▶ For example returning pointer to array of 10 int becomes:

```
1 auto func( int i ) -> int (*) [10] ;
```

Using decltype

- ▶ Another alternative is to use `decltype`
- ▶ For example returning pointer to array of odd or even integers:

```
1 int odd [] = {1, 3, 5, 7, 9};
2 int even [] = {0, 2, 4, 6, 8};
3 decltype( odd ) * oddsOrEvens( int i ){
4     return ( i%2 == 0 ) ? &even : & odd;
5 }
6 int main(){
7     int iin;
8     if ( cin >> iin ){
9         decltype(odd) * arr = oddsOrEvens( iin );
10        for ( auto i : *arr ){
11            cout << i << " ";
12        }
13        cout << endl;
14    }
15    return 0;
16 }
```

- ▶ See example `ptrToArrayFuncs.cpp`

Overloaded functions overview

- ▶ Functions having the same name in the same scope need to have different argument types, and are called **overloaded functions**
- ▶ For example overloaded `printarr` function to print array:

```
1 void printarr( const int * i1 );
2 void printarr( const int * beg, const int * end );
3 void printarr( const int arr [], const size_t n );
4 int main() {
5     int i = 10;
6     int array [] = { 96, 24, 6 };
7     printarr( &i );
8     printarr( begin( array ), end( array ) );
9     printarr( array, end(array) - begin(array) );
10    return 0;
11 }
```

- ▶ Note: `main` function may not be overloaded

Defining overloaded functions

- ▶ Consider particle physics event viewer with several functions to locate an **Event**, by a unique event number, by filename and record number, by date and time type **tm**:

```
1 Event getevent( const int EvNo );
2 Event getevent( const string& fname, const int
   entry );
3 Event getevent( const tm &adt );
4 int main(){
5     int i = 12345;  tm evtime;
6     // build time for 2012 Jun 18 16:05:03
7     evtime.tm_year=2012-1900; evtime.tm_mon=6;
8     evtime.tm_mday=5; evtime.tm_hour=16;
9     evtime.tm_min=5; evtime.tm_sec=3;
10    evtime.tm_isdst = 0;
11    Event evt1 = getevent( 100 );
12    Event evt2 = getevent( "evens.dat", 1 );
13    Event evt3 = getevent( evtime );
14    evt1.Print();
15    evt2.Print();
16    evt3.Print();
17    return 0;
18 }
```

Determining whether two parameter types differ

- ▶ Two parameter lists can be identical even if they don't look the same
- ▶ The compiler will likely let you know if two parameter lists are the same

```
1 // each pair below declares same function
2 Event getevent( const tm &atime );
3 Event getevent( const tm& ); // par names ignored
4 typedef const tm EVTime;
5 // below const tm and EVTime are the same
6 Event getevent( const tm& );
7 Event getevent( EVTime& );
```

Overloading and const parameters

- ▶ Top-level const has no effect on objects passed to function
- ▶ Following re-declare same function:

```
1 Event getevent( tm );  
2 Event getevent( const tm ); // redeclares  
   getevent(tm)  
3 Event getevent( tm* );  
4 Event getevent( tm* const ); // redeclares  
   getevent(tm*)  
5 Event getevent( const tm *); // ok: new function  
6 Event getevent( tm& );  
7 Event getevent( const tm&); // ok: new function
```

Overloading and `const_cast`

- ▶ Useful way of having same behaviour of non-const reference parameters and const reference parameters is to use `const_cast`
- ▶ Consider our `shorterString` function:

```
1 const string &shorterString( const string &s1 ,
2                             const string &s2 ){
3     return s1.size() <= s2.size() ? s1 : s2;
4 }
```

- ▶ We can call above with non-const arguments, but we get back a reference to const string. May want to get back non-const:

```
1 string &shorterString( string &s1, string &s2 ){
2     auto retval = shorterString( const_cast<const
3                                 string&>( s1 ),
4                                 const_cast<const
5                                     string&>( s2 )
6                                     );
7     return const_cast<string&>( retval );
8 }
```

Calling an overloaded function

- ▶ For any call to an overloaded function there are three possible outcomes:
 - ▶ Compiler finds one function that **best matches** the arguments and generates code to call that function
 - ▶ There is no function with the parameters matching the arguments – in which case compiler will issue an error message
 - ▶ There is more than one function that matches and none of the matches is clearly the best – in this case the call is **ambiguous**

Overloading and scope

- ▶ Following example is a bad idea – but shows effect of scope of name when declaring overloaded function in local scope

```
1 void print( const string & );
2 void print( const double & );
3 int main(){
4     int ival;
5     string sval;
6     if ( cin >> ival >> sval ){
7         print("Value");
8         print( ival ); // ok: uses print(const double&)
9         void print( int ); // hides prev. instances of
            print
10        print( sval ); //error: print(const string&)
            hidden
11        print( ival ); //ok: print(int) is visible
12        print( 9.99 ); //ok: calls print(int)
13    }
14    return 0;
15 }
```

Default arguments

- ▶ We can declare commonly used parameter arguments
- ▶ For example perhaps we have a string to represent the contents of the screen
- ▶ Screen has a default size in characters wide and characters high, but could have other sized screens

```
1 typedef const string::size_type cst;
2 string screen( cst hh = 30, cst ww = 80, const
   char bg = ' ' );
```

- ▶ Every parameter after first one given a default value must also be given a default parameter
- ▶ Then can call function using 0, 1, 2 or 3 parameters.

```
1 string w1 = screen();
2 string w2 = screen(66);
3 string w3 = screen(77, 22);
4 string w4 = screen(88, 33, ' ');
```


Default argument initializers

- ▶ Can use any expression that has a type convertible to the type of the parameter
- ▶ Local variables cannot be used

```
1 // values declared outside function:
2 typedef string::size_type st;
3 st ww = 80;
4 st hh();
5 char bg = ' ';
6 string screen( st = ww, st =hh(), char = bg );
7 void myfunc() {
8     bg = '.';
9     // below hides outer def of ww,
10    // but doesn't change default value
11    sz ww = 100;
12    // below calls
13    // screen( hh(), 80, '.' );
14    st win = screen();
15 }
```

inline functions

- ▶ `inline` function is expanded “in line” at each call
- ▶ run-time overhead of calling function is removed
- ▶ `inline` is meant to optimize small single line functions
- ▶ For example the `shorterString` function is short enough

```
1 inline const string &  
2 shorterString( const string &s1,  
3               const string &s2 ){  
4     return s1.size() <= s2.size() ? s1 : s2;  
5 }
```

- ▶ `inline` specification is request to compiler (which could be ignored).

constexpr Functions

- ▶ Sometimes we want a function calculation to be done at compile time
- ▶ That is done particularly if the function is used a lot
- ▶ **constexpr function** is used to let the compiler know that we want this calculation done at compile time.
- ▶ **constexpr** function body must contain exactly one return statement
- ▶ examples:

```
1 constexpr double cdpi() { return 3.14159265359; }
2 constexpr double mypi = cdpi();
3 constexpr double npi( int n ){ return cdpi() * n }
```

- ▶ **constexpr** functions are implicitly **inline**
- ▶ **constexpr** function is not required to return a constant expression

assert preprocessor macro

- ▶ **assert** is a preprocessor macro in header file `<cassert>` that takes a single condition
- ▶ if the condition is **true** (non-zero) then **assert** writes a message and terminates the program.

```
1 assert ( value != 0 );
```

- ▶ **assert** macro can be used to check for conditions that would cause the program to fail

NDEBUG preprocessor variable

- ▶ Can use preprocessor check `#ifndef NDEBUG` to put in debug printouts
- ▶ Can turn off debugging by defining `#define NDEBUG`, or passing it as compiler option `g++ -D NDEBUG`
- ▶ Example debug printing:

```
1 void myfunc( const int arr [], size_t n ){
2     #ifndef NDEBUG
3         // __func__ is a local static
4         // defined by compiler that holds function's name
5         cerr << __func__ << " array size = " << n <<
6             endl;
7     #endif
8     // ...
9 }
```

- ▶ `__func__` is used to print function name
- ▶ Other preprocessor defines of use in debug printouts:
 - `__FILE__` string literal with name of file
 - `__LINE__` int literal with line number
 - `__TIME__` string literal with compile time
 - `__DATE__` string literal with compile date

Pointers to functions

- ▶ A function pointer can be defined to point to a particular type of function
- ▶ Consider the function:

```
1 const string & shorterString( const string & s1,  
    const string & s2 );
```

- ▶ We can declare a pointer to a function of this type:

```
1 // pf is an uninitialized pointer to function  
2 const string & (*pf)( const string &, const string  
    & );  
3 // point it to a function:  
4 pf = shorterString; // pf now points to  
    shorterString function  
5 pf = &shorterString; // equivalent to above (&  
    optional)  
6 // can call function  
7 const string & s;  
8 s = pf( "short", "long" ); // calls shorterString  
9 s = *pf( "long", "short" ); // equivalent to above
```

More using function pointers

- ▶ return type and arguments of function must match pointer definition
- ▶ can assign `nullptr` or zero to indicate pointer is not pointing to a function

```
1 string::size_type getSumLength( const string&,
2                                 const string& );
3 bool stringsEqual( const char*, const char* );
4 const string & (*pf)( const string &, const string
5                       & );
6 pf = nullptr; // ok: pf points to no function
7 pf = getSumLength; // error: return type differs
8 pf = stringsEqual; // error: param types differ
9 pf = shorterString; // ok: function and ptr types
10                       match
```

Pointers to overloaded functions

- ▶ Pointer definition makes it clear which of a set of overloaded functions it refers to
- ▶ for example

```
1 void func( int * );
2 void func( unsigned int );
3 void func( double );
4 // below we define a pointer to the function
5 // void func( unsigned int )
6 void (*pfunc1)( unsigned int ) = func;
7 // below are errors , as no function of type exists
8 void (*pfunc2)( int ) = func;
9 double (*pfunc3)( int* ) = func;
```


Function pointers in function parameters

- ▶ We can write a parameter that looks like a function – will be treated as a pointer
- ▶ for example following function declarations are equivalent

```
1 void printOnCondition( const string &s1, const
    string &s2,
2     const string & pf( const string & s1,
        const string & s2 ) );
3 void printOnCondition( const string &s1, const
    string &s2,
4     const string & (*pf)( const string & s1,
        const string & s2 ) );
```

- ▶ We can pass function as argument directly:

```
1 printOnCondition( "long", "longer", shorterString
    );
```

Returning a pointer to function

- ▶ Often writing function pointer types gets tedious, so use typedef to define type:

```
1 const string & shorterString( const string & s1,
    const string & s2 );
2 typedef const string & csr;
3 typedef const string & (* fpSS )(csr s1, csr & s2);
4 fpSS fp = shorterString;
5 void printOnCondition( csr , csr , fpSS );
6 // above equivalent to
7 void printOnCondition( const string &s1, const
    string &s2, const string & (*pf)( const string
    & s1, const string & s2 ) );
```

- ▶ See `funcFun.cpp` for another example

Function pointers in functions

- ▶ Often writing function pointer types gets tedious, so use typedef to define type
- ▶ Can also use decltype to simplify notation

```
1 const string & shorterString( const string & s1,  
    const string & s2 );  
2 typedef const string & csr;  
3 // following two lines are equivalent  
4 typedef const string & (* fpSS1 )( csr s1, csr &  
    s2 );  
5 typedef decltype( shorterString ) * fpSS2;  
6 fpSS fp = shorterString;  
7 void printOnCondition( csr , csr , fpSS2 );  
8 // above equivalent to  
9 void printOnCondition( const string &s1, const  
    string &s2,  
10     const string & (*pf)( const string & s1,  
    const string & s2 ) );
```

Returning a pointer to function

- ▶ Pointer to a function can be the return value of a function
- ▶ Use type alias:

```
1 using F = int ( int* , int ); // F is a function
   type
2 using PF = int (*) ( int*, int ); // PF is a
   pointer type
3 PF func1( int ); // ok: func1 returns pointer to
   function
4 F * func2( int ); // ok: func2 returns pointer to
   function
5 F func3( int ); // error: cant return function
```

Returning a pointer to function

- ▶ Directly declaring function can be done without alias as well:

```
1 int ( *func4(int) ) ( int *, int );  
2 // alternatively use trailing return notation  
3 auto func4( int ) -> int (*) ( int*, int );
```

- ▶ Need to read first one above from the inside out:
 - ▶ func4 has parameter list, so it is a function
 - ▶ it is preceded by a * so it returns a pointer
 - ▶ the type of pointer has parameter list, so it points to a function that returns an int

Outline

Functions Part II

classes

- Introduction to classes

- Interface and implementation

- Enumerations

- Operator overloading

- Class interfaces

Introduction to classes

- ▶ We can define our own data type structures using **struct**
- ▶ To extend this idea to have definitions of objects that contain a particular data structure and ways of acting on them we can use **class**
- ▶ Fundamental ideas are to provide **abstract data types** and **encapsulation**
- ▶ An abstract data type:
 - ▶ class designer worries about how class is implemented (the hard work)
 - ▶ class user has access to the interface but not to the implementation (doesn't have to know how it works, just what its effect is)
- ▶ Class designer needs to implement what the class does
- ▶ Class user just needs to know class interface (how to use it)

What is a class

- ▶ Want to represent some idea or concept in code using data structures plus a set of functions
- ▶ A class represents a concept in a program.
- ▶ It knows how to represent the data needed in an object
- ▶ It knows what operations can be applied to those objects
- ▶ A class is a user-defined type that specifies how objects of its type are represented
 - ▶ How they are created
 - ▶ How they are used
 - ▶ How they are destroyed
- ▶ Examples of possible classes: vector, matrix, input stream, string, FFT (fast fourier transform), valve controller, robot arm, picture on screen, dialog box, graph, window, temperature reading, clock, ...

Defining a class

- ▶ A class is composed of built-in types and other user defined types
- ▶ Parts used to define the class are called **members**, for example:

```
1 class XX {
2     public:
3         int x; // data member
4         // below is member function
5         // set x to ax return old value of x
6         int setx( int ax){
7             int old=x; x=ax;
8             return old;
9         }
10 };
```

- ▶ We access members using `object.member` notation:

```
1 XX myx;
2 myx.x = 4; // set value of x in myx
3 // below: call member function setx
4 int locx = myx.setx( 9 );
```

class Interface and Implementation

- ▶ Interface is part that users accesses and is identified with the label **public**:
- ▶ Implementation has details of calculations and encapsulates the data and is identified with the label **private**

```
1 class A { // class's name is A
2     public:
3         // public members:
4         // - interface to users accessible by all
5         // - public functions (Get, Set, other...)
6         // - types
7         // - data -- often best kept private
8     private:
9         // private members:
10        // - implementation details (members only)
11        // - functions
12        // - types
13        // - data
14 };
```

- ▶ Note that class members are private by default, unless they are put below a **public**: tag.

Private members

- ▶ Private members cannot be accessed by the user directly
- ▶ Instead define public functions to access private members

```
1 class B {
2     int a;
3     int mult2(){ a*=2; return a; }
4     public:
5     void set( int aa ){ a=aa; }
6     int get(){ return mult2(); }
7 };
8 int main(){
9     B b; // variable b of type B
10    b.a = 2; // error: b.a is private
11    b.set( 2 ); // ok: sets b.a to 2
12    int c = b.mult2(); // error: mult2 is private
13    int d = b.get(); // ok: d and b.a are now 4
14    return 0;
15 }
```

class and struct ¹

- ▶ A **struct** is a **class** whose members are **all** public.
- ▶ **struct** typically used when only data needs to be represented and any value of the members of the struct are valid
- ▶ Note that **struct** can have member functions and constructors defined, in a similar way as **class**
- ▶ Consider a simple **struct** to represent a date:

```
1 struct Date {  
2     int y; // year  
3     int m; // month  
4     int d; // day  
5 };  
6 // Define a Date}variable named today  
7 Date today;
```

- ▶ **Date** object is simply collection of three **int** values

¹These notes and the Date example are based on B. Sroustrup, "Programming principles and practice using C++ ," Ch.9, 2014.

Problems with the Date struct

- ▶ We can do just about anything we want with `Date` objects.
- ▶ Everything is tedious and error-prone however:

```
1 // set today to October 31, 2016
2 Date today;
3 today.y = 2016; today.m=10; today.d=31;
4 // set a clearly nonsense date:
5 Date x;
6 x.y = -200; x.m=24; x.d = 100;
7 // set a date (but was there a leapyear?)
8 Date leap;
9 leap.y=2000; leap.m = 2; leap.d=29;
```

- ▶ We can define helper functions to initialize date and do operations, such as adding `n` days (maybe wrapping month and/or year)

```
1 bool init_day( Date & dd, int y, int m, int d){
2     // check that y, m, d is a valid date
3     // if it is, initialize dd and return true
4     // otherwise return false
5 }
6 void add_day( Date & dd, int n ){
7     // increase Date by n days
8 }
```

Using Date struct

- ▶ Now try using Date struct:

```
1 void func() {
2     Date today;
3     // oops below returns false to ok
4     // since order should have been y, m, d
5     bool ok = init_day( today, 12, 24, 2003);
6     add_days( today, 10 ); // add 10 days
7 }
```

- ▶ Clearly checking values in initialization is useful
- ▶ When defining user types want to define operations on it so that validity of result can be checked
- ▶ Need to ask ourselves: “What operations do we want for this type?”

Member functions and constructors

- ▶ Providing checking functions is good, but are of little use if we fail to call them
- ▶ For example, suppose we defined output operator for `Date`

```
1   Date today;  
2   // ...  
3   cout << today << endl;  
4   // ...  
5   init_day( today , 2013, 12, 31 );  
6   // ...  
7   Date tomorrow;  
8   tomorrow.y = today.y;  
9   tomorrow.m = today.m;  
10  tomorrow.d = today.d + 1;  
11  cout << tomorrow <<endl;
```

- ▶ We forgot to initialize `today` before using it
- ▶ We also forgot to use `add_day` function to add a day, and therefore end up with an invalid date in `tomorrow`
- ▶ To fix these bad code problems from happening, define member functions and constructors.

Member functions and constructors

- ▶ A member function with the same name as its class is called a **constructor**
- ▶ The constructor is used to initialize an object of that class

```
1 struct Date {
2     int y, m, d; //year, month, day
3     Date( int y, int m, int d ); // check valid date
4     void add_days( int n );
5 };
6 //...
7 Date birthday; // error: birthday not initialized
8 Date today{ 12, 24, 2007 }; // run-time error
9 Date xmas{ 2016, 12, 25 }; // correct!
10 Date next = { 2017, 12, 25 }; // ok, more verbose
11 Date prev = Date{ 2015, 12, 25 }; // ok (verbose)
12 Date old = Date( 1998, 12, 25 ); // ok old style.
13 xmax.add_days(2); // ok
14 add_day(2); // error: add to what date?
```


Remaining problem to solve...

- ▶ Still not required to use `add_day()` method of structure.
- ▶ Also, can still directly change `y`, `m`, or `d` in the `Date` struct, possibly making values invalid
- ▶ Solution is to make data private – requires class
- ▶ Private data **encapsulates** it so that it can only be changed by calling member functions that check validity of changes

```
1 class Date {
2     int y, m, d; // year, month, day
3     public:
4         Date( int ay, int am, int ad ); // check
           validity
5         void add_days( int n ); // increase date by n
           days
6         int month() { return m; }
7         int day() { return d; }
8         int year() { return y; }
9     }; //...
10 Date bday{ 2003, 12, 24 }; //ok
11 bay.m = 14; // error: Date::m is private
12 cout << bday.month() << endl; // ok: access month
```

Valid values

- ▶ Constructor of class ensures only valid objects are created
- ▶ Design classes so constructor only creates valid objects
- ▶ Value of object is called its **state**
- ▶ A rule for what constitutes a valid value is called an **invariant**
- ▶ The **invariant** for **Date** is a bit tricky to get right
- ▶ Need to worry about leap years and time zone (and assume a particular calendar – ie not Gregorian, Julian or Mayan calendar)
- ▶ If no good invariant exists, probably best to use plain data in a **struct**

Date class declaration

- ▶ So far designed interface to `Date` class, eventually need to implement member functions
- ▶ First, here is more common ordering of putting public interface first:

```
class Date {
public:
    Date( int y, int m, int d );
    void add_days (int n );
    int month();
    // ...
private:
    int y, m, d; // year, month, day
};
```

- ▶ Putting interface first makes it easier for users of the class to find how to use it, without having to look at private part
- ▶ Declaration of class above should appear in header file
- ▶ Implementation of class should appear in `cpp` file of same name.

Defining member functions

- ▶ When defining member functions need to say which class it is a member of using notation `class_name::member_name`:

```
1 Date::Date( int ay, int am, int ad )
2   : y{ ay }, m{ am }, d{ ad } {
3   // constructor -- note member initializers above
4   // now check validity of date ...
5 }
6
7 void Date::add_day( int n ){
8   // ... add to days to date
9 }
10
11 // note other one liner methods could be defined
12 // in class declaration
13 int Date::month(){
14   return m;
15 }
```

- ▶ Note the `:t{ay}, m{am}, d{ad}` notation for initializing members (member initializer list)
- ▶ In general – don't put member function implementation in class declaration to keep declaration shorter and easier to read

Example use of Date objects

- ▶ Each object of a particular type keeps its own set of data
- ▶ Member function calls for particular object acts only on data of that object

```
1 void PrintTwoDates( Date d1, Date d2 ){
2     cout << " Date 1: " << d1.year() << "/"
3         << d1.month() << "/" << d1.day();
4     cout << " Date 2: " << d2.year() << "/"
5         << d2.month() << "/" << d2.day();
6     cout << endl;
7     return;
8 }
9 // ...
10 Date today{2016,10,13};
11 Date tomorrow{2016,10,14};
12 PrintTwoDates( today, tomorrow );
```

Reporting errors

- ▶ Q: What do we do if we get a bad Date initialization?
- ▶ A: Throw an exception!
- ▶ We can put error checking in separate member function since it has different function than constructor

```
1 class Date{
2     public:
3         class Invalid {}; // to be used as exception
4         Date( int y, int m, int d );
5         //...
6     private:
7         int y, m, d; // year, month, day
8         bool is_valid(); // return true if date is
9             valid
10        bool is_leapyear(); // return true for leapyer
11 };
12 // then in cpp file:
13 Date::Date( int ay, int am, int ad )
14     : y( ay ), m( am ), d( ad ) {
15     if ( ! is_valid() ) throw Invalid();
16 }
```

Example use of Date exception

► Example use of date exception

```
1 bool DateValid(int d, int m, int y){
2     try {
3         Date adate{ y, m, d };
4         // assuming we define << operator for Date
           objects:
5         cout << adate << " is valid" <<endl;
6     } catch ( Date::Invalid ) {
7         cerr << "Invalid date" << endl;
8         return false;
9     }
10    return true;
11 }
```

Enumerations

- ▶ `enum` is a simple user defined type that specifies a set of values as symbolic constants
- ▶ Example enumerations:

```
1 enum class Month {
2     jan=1, feb , mar, apr , may, jun , jul , aug, sep ,
3     oct , nov, dec
4 };
5 // below monday is represented by 0
6 enum class Day {
7     monday, tuesday , wednesday , thursday , friday ,
8     saturday , sunday
9 };
10 // We can use enums as follows
11 Month m1 = Month::feb;
12 Month m2 = feb; // error: feb not in scope
13 m1=7; // error: can't assign an int to Month
14 int n=m1; // error: can't assign a Month to an int
15 Month mm = Month( 7 ); // convert int to Month
16     (unchecked)
```


Enumerations

- ▶ Nothing prevents someone from writing `Month m(9999)`
- ▶ Can write functions to check validity of `enum` values

```
1 Month int_to_month( int x ){
2     if ( x < int( Month::jan ) ||
3         x > int( Month::dec ) ) throw
4         runtime_error( "bad month" );
5     return Month(x);
6 }
```

- ▶ Enum is useful whenever a set of related integer constants is used
- ▶ Examples: (red, blue, green, ...), (yes, no, maybe), ...

Plain enumerations

- ▶ `enum class` defined on previous slides also called a scoped enumeration
- ▶ `enum class` is new to C++ and should be preferred over plain `enum`
- ▶ `enum` without `class` keyword are plain enumerations that export all of their enumerators to the scope of the `enum` definition
- ▶ Example use:

```
1 enum Month {  
2     jan=1, feb, mar, apr, may, jun, jul, aug, sep,  
3     oct, nov, dec  
4 };  
5 Month m = feb; // okay: feb in scope  
6 Month m2 = Month::feb; // okay too.  
7 m = 7; // error: can't assign an int to Month  
8 int n = m; // ok: can assign Month to int  
9 Month m3 = Month(7); // convert int to Month  
10      (unchecked)
```

Operator overloading

- ▶ C++ operators can be defined for class or enum operands
- ▶ First Example for our Month enum:

```
1 enum class Month {
2     Jan=1, Feb, Mar, Apr, May, Jun, Jul,
3     Aug, Sep, Oct, Nov, Dec };
4 //overload prefix increment operator
5 Month operator++( Month& m ) {
6     // Logic to wrap around month
7     m = (m==Month::Dec) ? Month::Jan : Month(
8         int(m)+1 );
9     return m;
10 }
11 // ...
12 Month m = Month::Sep;
13 ++m; // m is Month::Oct
14 ++m; // m is Month::Dec
15 ++m; // m is Month::Jan (wrapped around)
```

Operator overloading

- ▶ Probably more useful would be to define output stream operator:

```
1 const vector<const string> month_tbl = {  
2     "January", "February", "March", "April", "May", "June",  
3     "July", "August", "September", "October", "November",  
4     "December" };  
5 ostream& operator<<( ostream & os, Month m ) {  
6     return os << month_tbl[ unsigned(m)-1 ] ;  
7 }
```

- ▶ Could also define any operator provided by C++
- ▶ Such as +, -, *, /, %, [], (), ^, !, &, <, <=, >, >=
- ▶ Overloaded operator must have at least one user-defined type as operand
- ▶ Only overload operators in a way that uses their conventional meaning

Class interfaces

Key principles in designing a good interface:

- ▶ Make the interface complete.
- ▶ Make the interface minimal.
- ▶ Support copying or prohibit it (more later)
- ▶ Use types to provide good argument checking
- ▶ Identify non-modifying member functions
- ▶ Free all resources in the destructor

Argument types

- ▶ When we defined `Date` it had three `int` types as arguments
- ▶ Easy for user to make a mistake and supply y,m,d in wrong order (different conventions in different countries)
- ▶ One solution is to use `Month` type for month.
- ▶ Add simple `Day` and `Year` types as well:

```
1 class Year {
2     public:
3         Year( int x ) : y( x ) { ; }
4         int year() { return y; }
5     private:
6         int y; //assume year could be any int
7 };
```

Argument types continued

- ▶ And now Day type:

```
1 class Day {
2     static const int min = 1;
3     static const int max = 31;
4     public:
5         class Invalid {};
6         Day( int x ) : d{x} {
7             if (x<min || x>max) throw Invalid {};
8         }
9         int day() { return d; }
10    private:
11        int d;
12};
```

- ▶ Note that in Day we used `static const` for constants – keyword `static` means there will only be one copy of these constants across all instances of the class

Using Argument Types for class interface

- ▶ Now we can use `Day`, `Month`, and `Year` as types when defining our `Date`

```
1 class Date {
2     public:
3         Date( Year ay, Month am, Day ad );
4         //...
5     private:
6         Year  y;
7         Month m;
8         Day   d;
9 };
10 // Now using Date class becomes:
11 Date d1{ Year{2003}, Month::Dec, Day{24} }; //OK
12 // error below: day and month in wrong order
13 Date d2{ Year{2003}, Day{24}, Month::Dec };
14 // runtime error Day::Invalid below:
15 Date d3{ Year{2003}, Month::Nov, Day{32} };
```


Copying objects

- ▶ Can we copy our objects?
- ▶ If you don't say anything else default case is to copy members into new object
- ▶ For example

```
1 Date bday{ Year{2003}, Month::Dec, Day{24} };
2 Date holiday = bday; // copy bday into holiday
3 ++holiday; //increment day by one
4 Date hlwn = Date{Year{2106}, Month::Oct, Day{31}};
5 cout << Date( Year{2016}, Month::Oct, Day{14} )
6     << endl;
```

- ▶ For more complicated objects (particularly ones where we allocate memory or other resources) may need to implement your own copy constructor of form:

```
1 class Date {
2     public:
3         //...
4         Date( const Date & adate ); // copy constructor
```

Default constructors

- ▶ Programmer must supply arguments to our `Date` class
- ▶ There may be times when it would be useful to start with an empty date, and fill it later
- ▶ Need to supply a default constructor, but need to pick a default date (how about Jan. 1, 0000)?

```
1 class Date{
2     public:
3         // default constructor
4         Date() : y{Year{0}}, m{Month::Jan}, d{Day{1}} {}
5         // constructor for first day of a year
6         Date( Year ay ) : y{Year{ay}} { }
7         // ...
8     private:
9         // another alternative: provide defaults
10        // above default constructor would be:
11        // Date() {};
12        Year y{ Year{0} } ;
13        Month m{ Month::Jan } ;
14        Day d{ Day{1} } ;
15};
```

Using default constructors

- ▶ Now that default constructor exists, we can use it for example to setup a vector of dates

```
1 // below: ten elements with default date
2 vector< Date > holidays(10);
3 // now lookup dates and set each element of vector
4 // Without default constructor would have code:
5 const Date& default_date() {
6     static Date dd{ Year{0}, Month::Jan, Day{1} };
7     return dd;
8 }
9 vector< Date > holidays2( 10, default_date() );
```

- ▶ Again note use of **static** so that only one copy of default date in **dd** is created across all calls

const member functions

- ▶ Some variables aren't meant to be changed by a member function, so we call them `const`
- ▶ Sometimes we want to know if member function will modify the object – if not we also call it `const`
- ▶ `const` members can be called for `const` objects

```
1 // example of first helper function to compare year
2 // of two dates, shouldn't change the dates
3 bool sameYear( const Date & d1, const Date & d2 ){
4     ++d1; // error: trying to add a day to const Date
5     return d1.year() == d2.year() ;
6 }
7 // example: methods day, month year are const
8 // methods
9 class Date{
10     public: // ...
11     Day    day()    const; // cant modify object
12     Month  month() const;
13     Year   year()  const;
14     void   add_day( int n ); //can modify object
15     // ...
16 };
```

Use of `const` member functions

- ▶ Note that use of `const` member functions is possible for objects that are declared `const`
- ▶ Use of non-`const` member functions is not possible for objects that are declared `const`
- ▶ Examples:

```
1 // ...
2 Date d1{ Year{2003}, Month::Dec, Day{24} };
3 const Date d2{ Year{2016}, Month::Oct, Day{14} };
4 cout << "d1 = " << d1.day() << " d2=" << d2.day()
      << endl; //ok
5 d1.add_day( 5 ); // okay
6 d2.add_day( 1 ); // error: d2 is const
```

Members and helper functions

- ▶ Member functions are special in that they can modify the object
- ▶ If something goes wrong with object first place to look is at the Member functions
- ▶ If there are lots of Member functions – lots to debug
- ▶ Can instead have non-essential helper functions outside of the class that have to call the member functions to modify the object
- ▶ Also if we need to change implementation of Date only the member functions need to be updated, not the helper functions
- ▶ For example if we had started with Day as an int in our implementation of Date and wanted to update it
- ▶ Helper functions wouldn't need update such as:

```
1 Date next_Saturday( const Date & d ){
2     // access d using d.day(), d.month(), and
3     // d.year()
4     // make new date and return
5     // ... no need to update this when Date
6     // internals change
7 }
```

More on helper functions

- ▶ Helper functions typically take argument that are of the class type
- ▶ Often we use `namespace` to group helper functions with their class
- ▶ For example put `Date` and its helper functions in `Chrono` namespace:

```
namespace Chrono {
    enum class Month { /*...*/ };
    class Year { /*...*/ };
    class Day { /*...*/ };
    Date next_holiday( const Date& d );
    bool isLeapyear( const Year& y );
    bool is_valid_date( int y, int m, int d );
    bool operator==( const Date &d1, const Date &d2
        );
    // ...
    class Date {
        public:
            //...
    };
}
```

Date helper functions validity check

```
1 bool Chrono::is_valid_date(int y, int m, int d){
2     if ( m < int(Chrono::Month::Jan) ||
3         m > int(Chrono::Month::Dec) ) return false;
4     if ( d < 1 ) return false;
5     switch ( m ) {
6         case 1: case 3: case 5: case 7:
7         case 8: case 10: case 12:
8             if ( d > 31 ) return false; break;
9         case 2:
10            if (isLeapyear(Chrono::Year{y}) && d>29){
11                return false;
12            } else {
13                if ( d > 28 ) {
14                    return false;
15                }
16            }
17            break;
18        default:
19            if ( d>30 ) return false; break;
20    }
21    return true;
22 }
```


Have a look at Chrono

- ▶ Not all of the features discussed tonight need to be used in the solution
- ▶ Sometimes having strongly typed objects makes the code longer and harder to get right
- ▶ Have a look at a fairly minimal implementation of `Chrono.h`, `Chrono.cpp` and example programs using the `Date` class

Week6 Done!

- ▶ Homework #4 due Fri Oct. 20.
- ▶ Homework #5 due Fri Oct. 27.
- ▶ Reminder – midterm is Oct. 24 (19:00) – after week 7 notes.
- ▶ Next class after midterm Wed. Nov. 1 at 6 pm?