Lecture Material

- Declarations and definitions of classes in C++
- Member functions and variables
- Constructors
- The this pointer
- Destructor
- Function and operator overloading
- Inline functions

Interface for a Simple *Date* Class

- Here’s a simple class type declaration (not definition):

```
class DateType {
public:
  // constructor
  DateType();
  DateType(int newMonth, int newDay, int newYear);
  // accessor methods (get methods)
  int GetYear( ) const;    // returns Year
  int GetMonth( ) const;   // returns Month
  int GetDay( ) const;     // returns Day
private:
  int Year;
  int Month;
  int Day;
};
```

•Does this declare a data type or a variable?
•What does public/private mean?
•What does const mean?
•What are the member variables?
•What are the member functions?
•What is missing?

Use of a Simple *Date* Class

- Here’s a way to use this class...

•How are the variables initialized?
•Explain the use of two constructors...
•Can I access member variables?
•Can I access member functions?
 (what’s the difference?)

```
class DateType {
public:
  // constructor
  DateType();
  DateType(int newMonth, int newDay, int newYear);
  // accessor methods (get methods)
  int GetYear( ) const;     // returns Year
  int GetMonth( ) const;    // returns Month
  int GetDay( ) const;      // returns Day
private:
  int Year;
  int Month;
  int Day;
};
```

```
DateType today(3, 4, 2004);
DateType tomorrow, someDay;

//can I do this?
  cout << today.Month;
//              how about
  cout << today.GetMonth();
```

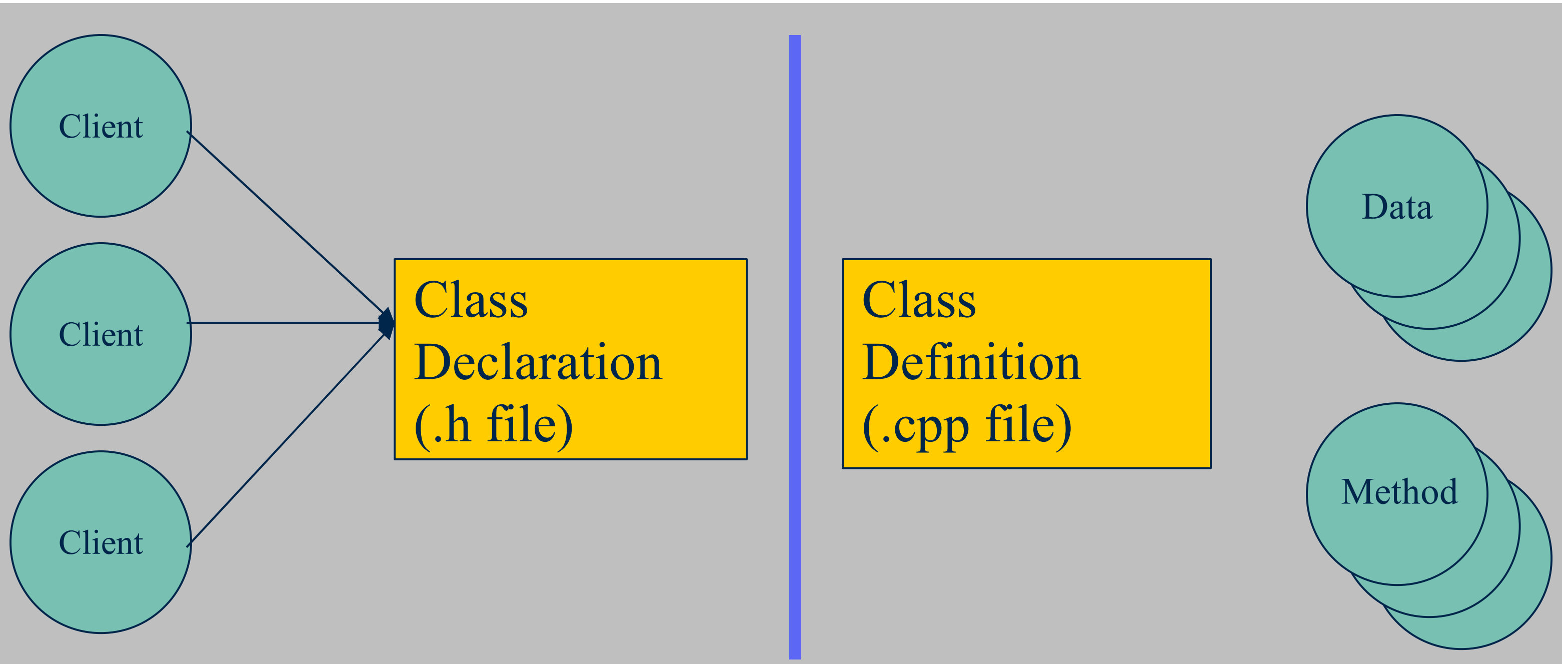
Implementation for a Simple *Date* Class

- Here’s the class type definition (aka implementation):

```
// DateType.cpp
#include "DateType.h"
/** Constructors **/
DateType::DateType() {
   Day = 1;
   Month = 1;
   Year = 1;
}
DateType::DateType(int newMonth, int newDay, int newYear) {
   Day = newDay;
   Month = newMonth;
   Year = newYear;
}
// returns Year
int DateType::GetYear( ) const { returns Year; }
// returns Month
int DateType::GetMonth( ) const { returns Month; }
// returns Day
int DateType:: GetDay( ) const { returns Day; }
```

- What was missing from the declaration?
- Where does the variables Day, Month, Year come from?
- Do we need the DateType.h to compile?

Encapsulation and Information Hiding



Client

Client

Client

Class Declaration (.h file)

Class Definition (.cpp file)

Data

Method

Clients Interface Implementation

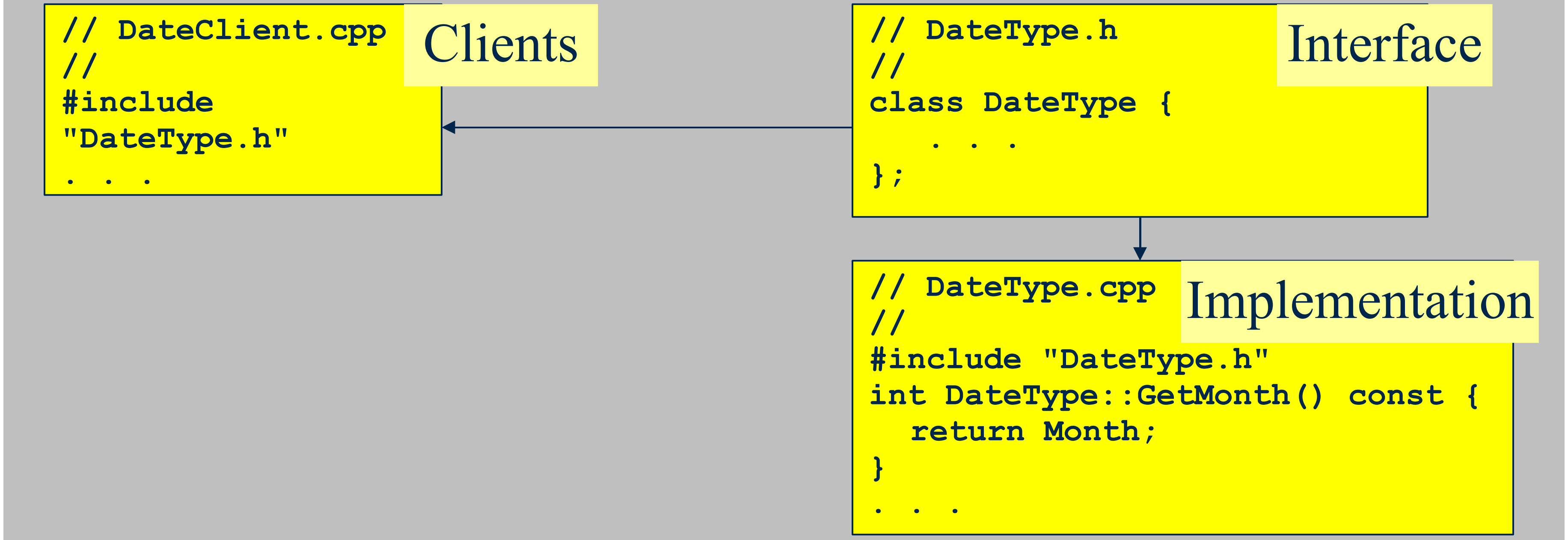
- Encapsulation
  - A C++ class provides a mechanism for bundling data and the operations that may be performed on that data into a single entity
- Information Hiding
  - A C++ class provides a mechanism for specifying access restrictions on both the data and the operations which it encapsulates

Implementation Organization

For separate compilation, a typical organization of the class implementation would involve two files:

`DateType.h`	class declaration
`DateType.cpp`	function member definitions

Suppose that a user of the DateType class writes a program consisting of a single source file, `DateClient.cpp`. Then the user would incorporate the DateType class files as follows:



```
// DateClient.cpp
//
#include
"DateType.h"
. . .
```

Clients

```
// DateType.h
//
class DateType {
   . . .
};
```

Interface

```
// DateType.cpp
//
#include "DateType.h"
int DateType::GetMonth() const {
  return Month;
}
. . .
```

Implementation

Using the Member Functions

In addition to using the class member functions directly, a client of a class may also implement higher-level functions that make use of the member functions. For example:

```
enum RelationType {Precedes, Same, Follows};
RelationType ComparedTo(DateType dateA, DateType dateB) {
  if (dateA.GetYear() < dateB.GetYear())
      return Precedes;
  if (dateA.GetYear() > dateB.GetYear())
      return Follows;
  if (dateA.GetMonth() < dateB.GetMonth())
      return Precedes;
  if (dateA.GetMonth() > dateB.GetMonth())
      return Follows;
  if (dateA.GetDay() < dateB.GetDay())
      return Precedes;
  if (dateA.GetDay() > dateB.GetDay())
      return Follows;
  return Same;
}
```

Client

Using the Member Functions

Then:

```
DateType Tomorrow(1,18,2002), AnotherDay(10, 12, 1885);
if ( ComparedTo(Tomorrow, AnotherDay) == Same ) {
  cout << "what do you think?" << endl;
}
```

Of course, the DateType class designer could also have implemented a member function for comparing two dates.

In fact, that would be more natural and more useful, since there is one natural way for the comparison to be (logically) defined.

Additional *Date*Type Methods

```
// add to DateType.h:
enum RelationType {Precedes, Same, Follows}; // file scoped
RelationType ComparedTo(DateType dateA); // to public section
```

```
// add implementation to DateType.cpp:
RelationType DateType::ComparedTo(DateType otherDate) {
  if (Year < otherDate.Year)
      return Precedes;
  if (Year > otherDate.Year)
      return Follows;
  if (Month < otherDate.Month)
      return Precedes;
  if (Month > otherDate.Month)
      return Follows;
  if (Day < otherDate.Day)
      return Precedes;
  if (Day > otherDate.Day)
      return Follows;
  return Same;
}
```

```
if ( Tomorrow.ComparedTo(AnotherDay) == Same )
  cout << “Think about it, Scarlett!” << endl;
```

Using the Methods

Another example:

```
void PrintDate(DateType aDate, ostream& Out) {
  PrintMonth( aDate.GetMonth( ), Out );
  Out << ' ' << aDate.GetDay( )
      << ", " << setw(4) << aDate.GetYear( ) << endl;
}
void PrintMonth(int Month, ostream& Out) {
  switch (Month) {
      case 1: Out << "January"; return;
      case 2: Out << "February"; return;
      . . .
      case 12: Out << "December"; return;
      default: Out << "Juvember";
  }
}
```

Then:

```
DateType LeapDay(2, 29, 2000);
PrintDate(LeapDay, cout);
```

will print:

```
February 29, 2000
```

Taxonomy of Member Functions

Member functions implement operations on objects. The types of operations available may be classified in a number of ways. Here is one common taxonomy:

Constructor an operation that creates a new instance of a class (i.e., an object)

Mutator an operation that changes the state of one, or more, of the data members of an object; a special kind is the Setter

Observer (reporter) an operation that reports the state of one or more of the data members of an object, without changing them

Also called Accessors, Getters

Iterator an operation that allows processing of all the components of a data structure sequentially

In the DateType class, DateType( ) is a constructor while GetYear( ), GetMonth( ) and GetDay( ) are accessors. DateType does not provide any iterators or mutators.

Default Constructor

The DateType class has a two explicit constructor member functions.

It is generally desirable to provide a default constructor, since that guarantees that any declaration of an object of that type must be initialized. :

```
DateType::DateType( ) {
   Month = Day = 1; // default date
   Year = 1980;
}
```

The "default" constructor is simply a constructor that takes no parameters.

Constructor Rules

- the name of the constructor member must be that of the class
- the constructor has no return value; **`void`** would be an error
- the default constructor is called automatically if an instance of the class is defined; if the constructor requires any parameters they must be listed after the variable name at the point of declaration

Other Constructors

The DateType class also has a nondefault constructor. This provides a user with the option of picking the date to be represented (essential since there are no mutator member functions - only way to initialize variable in this example).

```
DateType::DateType(int aMonth, int aDay, int aYear)
{
  if ( (aMonth >= 1 && aMonth <= 12)
       && (aDay >= 1) && (aYear >= 1) ) {
       Month = aMonth;
       Day = aDay;
       Year = aYear;
  }
  else {
       Month = Day = 1; // handling user error
       Year = 1980;
  }
}
```

The compiler determines which constructor is invoked by applying the rules for overloaded function names

When the constructor requires parameters they must be listed after the variable name at the point of declaration:

```
DateType aDate(10, 15, 2000);
DateType bDate(4, 0, 2005); // set to 1/1/1980
```

Default Constructor Use

- If you do not provide a constructor method, the compiler will automatically create a simple default constructor. This automatic default constructor:
  - takes no parameters
  - calls the default constructor for each data member that is an object of another class
  - provides no initialization for data members that are not objects
- Given the limitations of the automatic default constructor:

Always implement your own default constructor when you design a class!

The *this* Pointer

- Consider the following not so elegant piece of code...
  - Is it syntactically legal? That is, will it compile?
  - Is it semantically legal? That is will it run without producing some runtime error?
  - Does it do something useful?

```
DateType::DateType(int Month, int Day, int Year) {
  Month = Month;
  Day = Day;
  Year = Year;
}
```

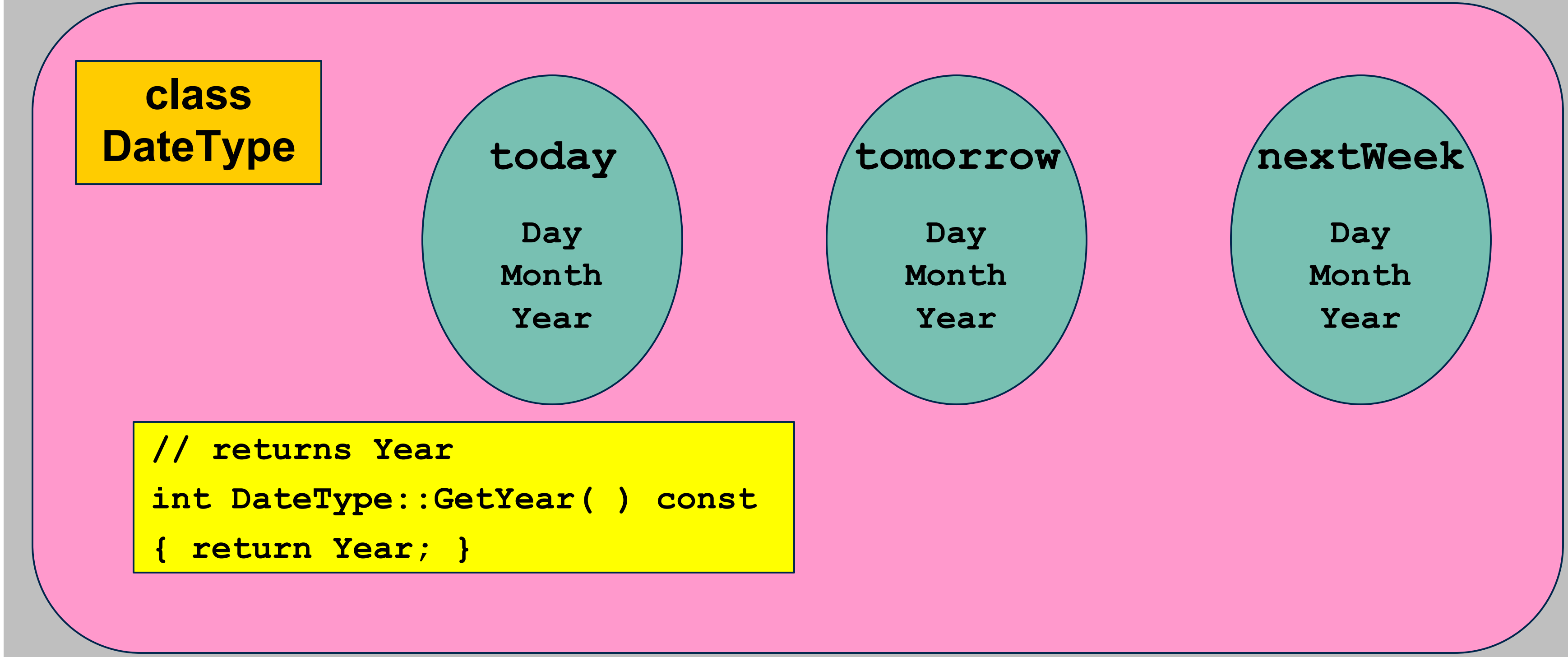
- For example, what is the variable today initialized to?

```
DateType today(3, 4, 2004);
```

The *this* Pointer

- You must think in terms of Classes and Objects to understand this

```
DateType today, tomorrow, nextWeek;
```



```
// returns Year
int DateType::GetYear( ) const
{ return Year; }
```

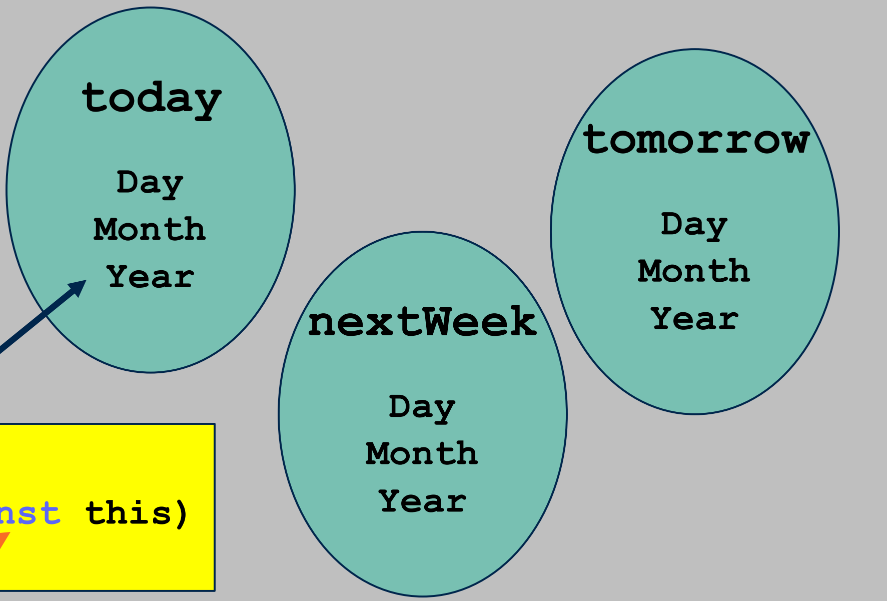
- In your class definition, how do you know which Day, Month or Year are you using?

The *this* Pointer

- C++ defines an additional implicit function parameter - the pointer to the current object instance: this

```
// returns Year
int DateType::GetYear( ) const
{ return Year; }
```

```
int y=today.GetYear();
```



```
// returns Year
int DateType::GetYear(const DateType* const this)
{ return this->Year; }
```

- This is how the compiler sees the method definition
- this is a constant pointer, you cannot modify it within a member function,
- Because the method is of const type (is an observer), this is also a pointer to a constant

The Answers Are

- Is it syntactically legal? Yes, it compiles
- Is it semantically legal? Yes, it runs fine
- Does it do something useful? No, it is a programmer error
- The solution would be:

```
DateType::DateType(int Month, int Day, int Year) {
  this->Month = Month;
  this->Day = Day;
  this->Year = Year;
}
```

Destructor

- Invoked automatically, when the variable is removed from memory (e.g. goes out of scope).
- Each class can have at most one destructor
- The destructor name is a name of a class preceded by a tilde sign (~).
- Destructor, the same as constructor, has no return type (even void)
- Destructor frees the resources used by the object (allocated memory, file descriptors, semaphores etc.)

```
// stack.h
class stack {
   ...
   int* data;
   ...
public:
   ...
   ~stack();
   ...
};
```

Interface

```
// stack.cpp
stack::~stack()
{
   free(data);
};
```

Implementation

```
{
   stack s;
   ...
   //s.~stack() invoked here
}
```

Client

Class *Stack*

```
//stack.h
#define STACKSIZE 20

class stack
{
public:
   void push(int a);
   int pop();
   void clear();
   stack();
   ~stack();
private:
   int top;
   int data[STACKSIZE];
};
```

Interface

```
//stack.cpp
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

stack::stack()
{
   this->top=0;
};
stack::~stack(){};
void stack::clear()
{
   this->top=0;
};
void stack::push(int a)
{
   assert(this->top<STACKSIZE);
   this->data[this->top++]=a;
};
int stack::pop()
{
   assert(this->top>0);
   return this->data[--this->top];
};
```

Implementation

Class *Stack* - An Improved Version

- Implement:
  - Dynamic memory allocation
  - Nonempty destructor

```
//stack.h
class stack
{
public:
   void push(int a);
   int pop();
   void clear();
   stack();
   ~stack();
private:
   int top;
   int *data;
   int size;
};
```

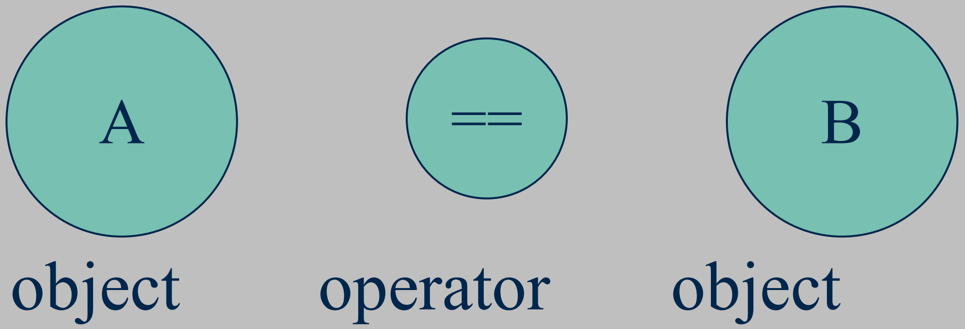
Interface

Overloading, Briefly

- In C++ it is legal to declare two or more functions with the same name. This is called overloading.
- The compiler determines which definition is referred to by each call
- The criteria used is (in the order listed):
  Considering types of the actual and formal parameters:
  - Exact match (no conversions or only trivial ones like array name to pointer)
  - Match using promotions (bool to int; char to int; float to double, etc.)
  - Match using standard conversions (int to double; double to int; etc.)
  - Match using user-defined conversions (not covered yet)
  - Match using the ellipsis . . . in a function declaration (ditto)
- The return type is not considered in overloading
- Keep this simple for now. Only overload a function name if you want two or more logically similar functions, like the constructors on the previous slides, and then only if the parameter lists involve different numbers of parameters or at least significantly different types of parameters.

Operator Overloading

- C++ language operators, (e.g., “==”, “++”, etc.) can be overloaded to operate upon user-defined types.



means

A.==(B)

- which means A’s type must have defined a member function of name = = with one argument

Operator Overloading

- Member functions overloading operators are defined using the keyword operator

```
// add to DateType.h:
bool operator==(DateType otherDate) const ;
```

Interface

```
// add to DateType.cpp:
bool dateType::operator==(DateType otherDate) const {
  return( (Day == otherDate.Day ) &&
  (Month == otherDate.Month ) &&
  (Year == otherDate.Year ));
}
```

Implementation

```
DateType aDate(10, 15, 2000);
DateType bDate(10, 15, 2001);
if (aDate == bDate) { . . .
```

Client

- When logically appropriate, overloading relational operators intelligently allows users to treat objects of a user-defined type as naturally as the simple built-in types.

Default Function Arguments

- Technique provided to allow formal parameters to be assigned default values that are used when the corresponding actual parameter is omitted.

```
// add to Datetype.h
DateType::DateType(int aMonth = 1, int aDay = 1, int aYear = 1980);
```

The default parameter values are provided as initializations in the parameter list in the function prototype, but not in its implementation.

```
// add to DateType.cpp
DateType::DateType(int aMonth, int aDay, int aYear)
{
  if ( (aMonth >= 1 && aMonth <= 12)
      && (aDay >= 1) && (aYear >= 1) ) {
      Month = aMonth;
      Day = aDay;
      Year = aYear;
  }
  else               {
      Month = Day = 1; // default date
      Year = 1980;
      }
}
```

This allows the omission of the default constructor, since default values are provided for all the parameters. In fact, including the default constructor now would result in a compile-time error.

Default Function Arguments

- If a default argument is omitted in the call, the compiler automatically inserts the default value in the call.

```
DateType dDate(2,29);     // Feb 29, 1980
DateType eDate(3);        // March 1, 1980
DateType fDate();         // Jan 1, 1980
```

- Restriction: omitted parameters in the call must be the rightmost parameters.

```
DateType dDate(,29);      // error
```

- Be very careful if you mix the use of overloading with the use of default function parameters.

Default parameter values may be used with any type of function, not just constructors. In fact, not just with member functions of a class.

Default Arguments Usage

- Default Arguments in prototypes
  - Omitted arguments in the function prototype must be the rightmost arguments.
- Default Arguments - Guidelines
  - Default arguments are specified in the first declaration/definition of the function, (i.e. the prototype).
  - Default argument values should be specified with constants.
  - In the parameter list in function declarations, all default arguments must be the rightmost arguments.
  - In calls to functions with > 1 default argument, all arguments following the first (omitted) default argument must also be omitted.
- Default Arguments and Constructors
  - Default argument constructors can replace the need for multiple constructors.
  - Default argument constructors ensure that no object will be created in an noninitialized state.
  - Constructors with completely defaulted parameter lists (can be invoked with no arguments) becomes the class default constructor (of which there can be only one).

Inline Member Functions

- Generally more efficient for small to medium functions
- Expanded in-place of invocation
  - Eliminates method invocation overhead
  - Compiler generates necessary code and maps parameters automatically
  - Still only one copy of function implementation (in case the programmer wants to take its address)
- Two ways to specify an inline method
  - Provide implementation during class definition (default inline)
  - Use ‘inline’ keyword in function definition (explicit inline)

Inline Member Function Examples

```
// DateType.h
class DateType {
public:
  DateType(int newMonth = 1, int newDay = 1,
  int newYear = 1980);
  int GetYear () const;
  int GetMonth ()const {return Month};
  int GetDay () const {return Day};
private:
  int Year, Month, Day;
};
```

Interface

```
// DateType.h
inline int DateType::GetYear() const { // explicit
inline
  return Year;
}
```

Implementation

Inline Member Functions

- Inline functions should be placed in header files to allow compiler to generate the function copies.
- Member functions defined in a class declaration are implicitly inlined.
- Efficiency is traded off at the expense of violating the information hiding by allowing the class clients to see the implementation.
- Reference to the class data members by the inline functions before their actual definition is perfectly acceptable due to the class scoping.

```
// DateType.h
class DateType {
public:
   DateType(int newMonth = 1, int newDay = 1,
   int newYear = 1980);
   int GetYear () const;
   int GetMonth ()const {return Month};
   int GetDay () const {return Day};
private:
   int Year, Month, Day;
};
```

Tradeoffs of Inline Methods

- Default inline violates some basic software engineering goals
  - separation of interface and implementation is broken
  - information hiding is lost – implementation is now exposed to clients
- All code that invokes an inline must be recompiled when:
  - method is changed
  - switching from inline to regular or vice-versa
- Inline is request, not command to compiler
  - Could manually inline, but at what cost…?
- Size of executable may increase
  - although that’s not usually that much of a factor