

CS481 - Object-Oriented Programming

Creating Manipulators

- ❖ It IS possible in C++ to create your own manipulators
- ❖ Zero-argument manipulators are easy to create:
 - function that takes an ostream&
 - returns an ostream&

1

Creating Manipulators

Declaration:

```
ostream& endl(ostream&);
```

Use:

```
cout << "hello world!" << endl;
```

2

Creating Manipulators

- ❖ endl produces a function address
- ❖ An applicator passes it a stream
 - via reference (&)
- ❖ Manipulators with arguments:
 - more complicated
 - effectors

3

Creating Manipulators

```
#include <iostream.h>
//defines the applicator

ostream& nl(ostream& os)
{
    return os << '\n';
} //like endl w/o flushing
```

4

Symbolic Constants

- ❖ C: #define macro directive
 - E.g., #define BUFSIZE 100
- ❖ C++: constant "variables"
 - E.g., const int bufsize = 100;
 - OK in compile-time constant expressions (e.g., char buf[bufsize])

5

const vs #define

- ❖ Avoids some macro ambiguities and errors
 - Extra space in declaration loses arguments
 - Unparenthesized argument errors
- ❖ Assigns a data type to value

6

CS481 - Object-Oriented Programming

Problems with #define

- ❖ #define sqr(x) x*x;
- ❖ sqr(y+1)
- ❖ expands to:
 - y + 1 * y + 1;
- ❖ yields unexpected result!!

7

const in Header Files

- ❖ const "variable" in .h file
 - Included by multiple files
- ❖ Duplicate definitions?
 - Will the linker complain?
- ❖ Default: internal linkage
 - Not visible outside source file

8

const in C vs C++

- ❖ In C:
 - always occupies storage
 - its name is global (*external* linkage)
 - cannot be used as a compile-time constant
 - ◆ can't do: const size = 100;
 - char buf[size];

9

const in C vs C++

- ❖ In C++:
 - rarely occupies storage
 - ◆ compiler holds value in symbol table
 - ◆ if pointed to, then storage is allocated
 - name is local to file (*internal* linkage)
 - can be used as a compile-time constant

10

Constant in Class

- ❖ Have global constants
 - Constant variable
 - Macro definition
- ❖ Constant class member?
 - Means constant for life of each object
 - Not one constant for entire class

11

Const Data Members

Interface	Implementation
<pre>class ConstEx { public: ConstEx(int ii); private: int i; const int ci; }</pre>	<pre>ConstEx::ConstEx(int ii) { // assignment i = ii; //OK ci = ii; //error! }</pre>

12

CS481 - Object-Oriented Programming

Const *Data Members*

Interface	Implementation
<pre>class ConstEx { public: ConstEx(int ii); private: int i; const int ci; }</pre>	<pre>ConstEx::ConstEx(int ii) { // assignment i = ii; //OK ci = ii; //error! }</pre> <p>Correct way to initialize ci is to use a <i>constructor initializer list</i>.</p>

13

Constant in Class

- ❖ Single class-wide constant?
 - E.g., a buffer-size parameter
- ❖ Possibilities:
 - Member: `static const` (??)
 - ◆ More on static later
 - Enum hack: `enum {size=10};`

14

Constant in Class

```
class Stack
{
  enum { size = 100 };
  int array[size]; ...
};
class Stack2
{ // more on "static" later
  static const int size;
  int array[size]; ...
};
const int Stack2::size=200;
```

15

const Type Modifier

- ❖ States intent not to modify
 - Compiler flags violations

16

const Type Modifier

- ❖ States intent not to modify
- ❖ Contexts
 - Simple variable - `const pi = 3.14;`
 - Pointer - `const int* ip;`
 - Data member - `const size;`
 - Object - `const Complex c1(1.0);`

17

const Objects

- ❖ Compiler can guarantee the const-ness of built-in types
- ❖ Cannot do so for user-defined types!
- ❖ Member functions can be made `const`

18

CS481 - Object-Oriented Programming

const Objects

- ❖ Can't modify object
- ❖ `const` member functions
 - E.g., `int GetValue() const;`
 - May not alter object members
 - ◆ Exception: Bit-wise vs member-wise??
(`mutable?`)

19

Casting Away Constness

- ❖ Pointers to `const` can be cast as pointers to non-const:
 - `((Stack*)this)->i++;`
- ❖ To be more visible to user code:
 - `mutable` keyword was added
 - not all compilers support this yet!

20

Mutable Keyword

- ❖ Appears in class declaration:


```
class Y
{
  int i;
  mutable int j;
public:
  Y() { i = j = 0; }
  void f() const;
};
```

21

Inline Functions

- ❖ Performance optimization
 - Function body inserted in-line
 - ◆ Instead of call to out-of-line function
- ❖ Specified in two ways:
 - `inline` keyword in function definition
 - Function body in class declaration

22

Inline Advantages

- ❖ Versus out-of-line
 - Better performance (simple functions)
 - ◆ But compiler hint only
- ❖ Versus preprocessor macro
 - Type safety (including `const`)
 - Avoids argument evaluation problems
 - proper scoping rules are followed. ²³

Access Functions

- ❖ Access to data members
 - Read/write, but not directly
 - Permit validation and future changes
- ❖ Safety at cost of performance?
 - Overhead of function call for access
- ❖ May be inline for efficiency! ²⁴

CS481 - Object-Oriented Programming

Access Functions

Inline function outside class declaration

Must be in a header file!

```
class rect
{ int width,height;
public: ...
  int Width() const;
  void Width(int w); ...
};
```

```
inline int rect::Width() const
{ return width }
```

25

Access Functions

Inline function in class declaration

```
class rect
{
  int width, height;
public:
  ...
  int Width() const
  { return width; }
  void Width(int w)
  { if (w>=0) width = w; }
};
```

26

Access Functions

"Set value" validation

```
class rect
{
  int width,height;
public:
  ...
  int Width() const
  { return width; }
  void Width(int w)
  { if (w>=0) width = w; }
};
```

27

Name Control

- ❖ Namespaces
 - namespace and using directives
 - ◆ Can supply alias for namespace
 - Not yet implemented?
- ❖ Static hack
 - Declare variable or function in class

28

static Keyword

- ❖ Static storage (same as C)
 - Static versus automatic variable
- ❖ Linkage (visibility) (same as C)
 - static means confined to file scope
- ❖ Static class members (new in C++)

29

Static Members

- ❖ Associated with entire class
 - Not with individual object (instance)
- ❖ Data member
 - A "global" variable in class scope
- ❖ Function member
 - No this pointer argument

30

CS481 - Object-Oriented Programming

Static Data Example

```
class Bunny
{
    static int how_many; ...
};
int Bunny::how_many = 0;
```

Maintain count in constructor & destructor

```
Bunny::Bunny()
{ how_many++; ... }

Bunny::~Bunny()
{ how_many--; ... }
```

Static Function

```
class Bunny
{ static int how_many; ...
public:
    static int HowMany()
    { return how_many; }
};
```

Invoke function with or without object

```
Bunny bunny1;
int n;
n = bunny1.HowMany();
n = Bunny::HowMany();
```

Static Name Scoping

- ❖ Can create a dummy class
 - With only static members
 - Never create object of the class
- ❖ Static data - pseudo-globals
- ❖ Static functions - utilities
 - Reference as `MyClass::FuncName`

