
CS11 – Introduction to C++

Fall 2009-2010

Lecture 3

Topics for Today

- C++ compilation process
- Using the `const` keyword
- Redefining operators for your classes



C++ Compilation

- You type:

```
g++ -Wall point.cc lab1.cc -o lab1
```

 - What happens?
 - C++ compilation is a multi-step process:
 - Preprocessing
 - Compilation
 - Linking
 - Different steps can produce different kinds of errors
 - ...very helpful to understand what is going on!
-

C++ Compilation: Overview

- For the preprocessing and compilation phases, each source file handled separately!

```
g++ -Wall point.cc lab1.cc -o lab1
```

- Compiler performs preprocessing and compilation on `point.cc` and `lab1.cc` separately
 - Produces `point.o` and `lab1.o`
 - The linking phase combines the results of the compilation phase
 - `point.o` and `lab1.o` are combined into a single executable program, called `lab1`
-

The Preprocessor

- Step 1: The Preprocessor
 - Prepares source files for compilation
 - Performs various text-processing operations on each source file:
 - Removes all comments from the source file
 - Handles preprocessor directives, such as `#include` and `#define`
 - Example: `lab1.cc: #include "point.hh"`
 - Preprocessor *removes* this line from `lab1.cc`, and *replaces* it with the contents of `point.hh`
-

The Preprocessor (2)

- Another common example: C-style constants
`#define MAX_WIDGETS 1000`
 - The preprocessor replaces all instances of `MAX_WIDGETS` with the specified text
 - After preprocessing, `MAX_WIDGETS` is gone, and source code contains `1000` instead
 - For each input source file:
 - (e.g. `lab1.cc`, `point.cc`)
 - The preprocessor generates a translation unit, i.e. the input that the compiler actually compiles
-

The Compiler

- The compiler takes a translation unit, and translates it from C++ code into machine code
 - i.e. from instructions that human beings understand, into instructions that your processor understands
 - Result is called an object file
 - e.g. `point.o`, `lab1.o`
 - These are not runnable programs, but they contain machine-code instructions from your program
-

The Compiler: Object Files

- Object files are incomplete! They specify, among other things:
 - Each function that is defined within the translation unit, along with its machine code
 - e.g. `point.o` contains a definition of:
`double Point::distanceTo(Point &p)`
 - This includes the function's instructions!
 - Each function that is referred to by the translation unit, but whose definition is not specified!
 - e.g. `lab1.o` uses `Point::distanceTo(Point &p)`, but doesn't include the definition of the function
-

The Linker

- The linker takes the object files generated by the compiler, and combines them together
 - Many object files refer to functions that they don't actually implement!
 - Linker makes sure that every function is defined in some object file
 - Two main kinds of errors:
 - Linker can't find the definition of a function
 - Linker finds multiple definitions of a function!
-

Linker Errors

- Example: you forget to include `main()`!
 - Example output on Mac OS X:
Undefined symbols:
 "`_main`", referenced from:
 start in crt1.10.5.o
ld: symbol(s) not found
 - `ld` is the linker program for `g++`
 - These errors don't occur during compilation!
 - Compilation has succeeded, but the linker can't find definitions for some functions
-

Final Compilation Notes

- Generally, compilers don't leave intermediate files around anymore
 - They use more efficient ways of passing translation units and object files to each other
 - Can compile a source file without linking it:
`g++ -Wall -c point.cc`
 - Performs preprocessing and compilation
 - Produces `point.o`
 - Can save other output of preprocessor, compiler:
`g++ -Wall --save-temps -c point.cc`
 - `point.i` is result of running the preprocessor
 - `point.s` is a text version of the processor instructions
-

Constants in C

- C constants typically specified with **#define**

- **#define** is a preprocessor directive

- Does simple text substitution

```
#define MAX_WIDGETS 1000
```

```
...
```

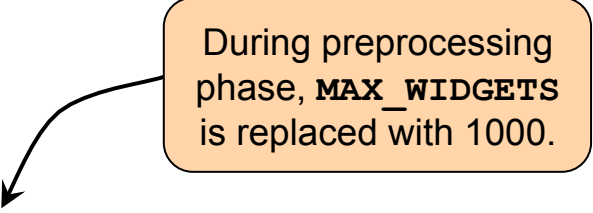
```
if (numWidgets > MAX_WIDGETS) ...
```

- **MAX_WIDGETS** has no type information!

- The compiler can't help you very much if you misuse it...

- e.g. `gcc` only gives warnings, but not errors

- In C++, the use of **#define** for constants is *strongly discouraged*.



During preprocessing phase, **MAX_WIDGETS** is replaced with 1000.

The **const** Keyword

- In C++, **const** is used for defining constants
 - const** `int MaxWidgets = 1000;`
 - Almost exactly like a normal variable declaration
 - **const** is a modifier – “this variable won’t change”
 - Now **MaxWidgets** also has type information
 - If you misuse it, the compiler will report errors.
-

C++ Constants

- Possibly slower than **#define**
 - May involve a variable lookup, depending on compiler implementation.
 - **const** is enforced at compile-time
 - All attempts to change **const** variables will result in compiler errors
 - ANSI C also supports **const** keyword
 - Wasn't in K&R C; was added in C99
-

Using **const** with Reference Arguments

- **const** is *very* useful with reference-arguments
 - “Don’t allow any changes to the referent!”
 - Yields the performance benefits, without the possibility of unintended side effects.
 - This is fast and dangerous:

```
double Point::distanceTo(Point &p) {...}
```
 - This is fast and safe:

```
double Point::distanceTo(const Point &p) {...}
```
 - Rule of thumb:
 - When you pass objects by reference, but you don’t want side effects, definitely use **const**.
-

Using **const** for Function Args

- Copy-constructors should use a **const**-reference.

```
Point(const Point &p); // Don't change the original!
```

- Efficient and safe

- Also use **const** with other functions that take object-references.

```
bool equals(const Point &p);  
double distanceTo(const Point &p);
```

- **const** has to appear in both function declaration *and* in function definition

- Constness is part of the argument's type
- If **const** declarations don't match, compiler thinks they are different functions!

```
bool equals(const Point &p)           (in Point.hh)  
bool Point::equals(Point &p)         (in Point.cc)
```

Next Question...

- Distance-to member function:

```
double Point::distanceTo(const Point &p) {  
    double dx = x_coord - p.getX();  
    double dy = y_coord - p.getY();  
    return sqrt(dx * dx + dy * dy);  
}
```

- How does compiler know that `getX()` and `getY()` don't change the other `Point`?
 - Compiler will complain if we don't say anything!
 - We tell the compiler in `Point`'s class-declaration, using `const`.
-

const Member Functions

- Specify **const** after function, when function doesn't change object's state.

```
class Point {  
    ...  
    double getX() const; // These don't change  
    double getY() const; // the Point's state.  
    ...  
};
```

- Must be specified in class declaration *and* in definition

```
double Point::getX() const {...}
```

- **distanceTo()** also doesn't change **Point** state

```
double Point::distanceTo(const Point &p) const  
{...}
```

const Return Values

- One other place you can specify **const**

- `const Point & getMyPointRef();`

- Means that the returned value can't change.
 - Or, in this case, the referent can't be changed.

- Also:

- `const Point getMyPoint();`

- Notice that this isn't a **Point**-reference
 - A useful trick, especially for operator overloads
-

C++ Operators!

- In C++, operators can be given new meanings

```
complex c1(3, 5);           // 3 + 5i
complex c2(-2, 4);         // -2 + 4i
complex c3 = c1 * 3 + c2;  // Do some math
```

- Called “operator overloading”
- “Syntax should follow semantics”
 - Looks less cluttered and more comprehensible.
 - Alternative:

```
complex c3 = c1;           // Do some math
c3.multiply(3);
c3.add(c2);
```

Operator Overloads

- You write: `c1 + c2`
 - The compiler sees: `c1.operator+(c2) ;`
 - `complex` defines a member function named `operator+`
 - This is a binary operator – it takes two operands.
 - Other binary operators follow similar pattern
 - Object on left-hand side (LHS) of operator gets called
 - Object on right-hand side (RHS) of operator is the argument
 - Example: `c1 = c2; // Assignment operator`
 - Compiler sees `c1.operator=(c2) ;`
 - Implement `complex::operator=()` member-function to support assignment for `complex`
-

The **FloatVector** Class

- Last week we looked at **FloatVector**
 - Variable-size vector of floating point numbers
- Declaration:

```
class FloatVector {
    int numElems;
    float *elems;

public:
    FloatVector(int n);
    FloatVector(const FloatVector &fv);
    ~FloatVector();
};
```

Assignment Operator

- = is the assignment operator

```
FloatVector fv1(30), fv2(30);
```

```
...
```

```
fv2 = fv1;    // Assign fv1 to fv2.
```

- **FloatVector** implements this function:

```
FloatVector & operator=(const FloatVector &fv);
```

- Argument is a **const** reference – it doesn't change
- Return-value is a non-**const** reference to the LHS of the assignment
- Allows operator chaining

```
// NOTE: = is right-associative, so rightmost = is  
//         evaluated first.
```

```
a = b = c = d = 0;
```

Implementing the Assignment Operator

- Assignment involves these steps:
 - Clean up current contents of object
 - (Just like the destructor does)
 - Copy the contents of the RHS into the object
 - (Just like the copy-constructor does)
 - Return a non-`const` reference to the LHS
 - `return *this;`
 - `this` is a new keyword in C++
 - When a member function is called on an object, `this` is a *pointer* to the called object
 - In `fv2.operator=(fv1)`, `this` is a pointer to `fv2`
 - Function returns what `this` points to: `return *this;`
-

Assignment Operations

- Assignment operator work overlaps with copy-constructor and destructor
 - Goal: Reuse code. Don't duplicate effort.
 - Factor common operations into helper functions

```
private:
    void copy(const FloatVector &fv);
    void cleanup();
```

 - Declared **private** so that only the class can use them
 - Side benefit: Only have to fix bugs in one place
 - Then, implement assignment, copy-ctor, destructor:
 - Copy-constructor calls **copy()**
 - Destructor calls **cleanup()**
 - Assignment calls **cleanup()** then **copy()**
-

Self-Assignment

- Always check for self-assignment in the assignment operator

```
FloatVector fv(100);  
fv = fv;    // Self-assignment!
```

- **fv** is on the left *and* right sides of the equals:

```
fv.operator=(fv);
```

- Step 1: Clean up contents of **fv** (the LHS)
 - Step 2: Copy contents of **fv** (RHS) into **fv** (LHS)
...but we just got rid of what **fv** contains
 - **fv** just self-destructed.
-

Checking for Self-Assignment

- Easy way to check for self-assignment:
 - Compare the addresses of LHS and RHS
 - If addresses are same, skip assignment operations
 - `this` is a pointer to the LHS, so that's easy
- Example:

```
FloatVector & FloatVector::operator=(const FloatVector &fv) {  
  
    // Make sure to avoid self-assignment.  
    if (this != &fv) {  
        ... // Do normal assignment operations.  
    }  
  
    return *this;  
}
```

Compound Assignment Operators

- C++ provides compound assignment operators:

- `+=` `-=` `*=` `/=` *etc.*
- Combines arithmetic operation and assignment

- These follow same pattern as simple assignment:

```
FloatVector & FloatVector::operator+=(const FloatVector &fv)
```

- Take a **const**-reference to RHS of operator
- Perform compound assignment operation
- Return a non-**const** reference to ***this**
 - Operator chaining works with compound assignment too
 - “If primitive types support it, user defined types should too”
- Should correctly handle self-assignment here too
 - (None of the C++ labs require you to care about this! 😊)

Simple Arithmetic Operators

- Pattern is slightly different for these:

```
const FloatVector FloatVector::operator+(  
    const FloatVector &fv) const;
```

- Take a **const**-reference to RHS, as usual
- Return a separate **const-object** containing result
 - Disallows stupid things like this:

```
fv1 + fv2 = fv3;
```

- **operator+** doesn't change object it's called on, so member function is declared **const** too
- Same for **- * /**
 - These produce a new object containing the result
-

Implementing Arithmetic Operators

- Simple arithmetic operators are *easy* once compound assignment operators are done

```
const FloatVector FloatVector::operator+(  
    const FloatVector &fv) const {
```

```
    // First, make a copy of myself.
```

```
    FloatVector result(*this);
```

```
    // Second, add the RHS to the result.
```

```
    result += fv;
```

```
    // All done!
```

```
    return result;
```

```
}
```

- Can combine into one line:

```
    return FloatVector(*this) += fv;
```

Equality Operators

- Equality `==` and inequality `!=` return `bool`

```
bool FloatVector::operator==(const FloatVector &fv) const;  
bool FloatVector::operator!=(const FloatVector &fv) const;
```

- RHS is a `const`-reference, as usual.
 - Member function is `const` too
 - Implementation guidelines:
 - Implement `==` first
 - Implement `!=` in terms of `==`

```
return !(*this == fv);
```
 - Ensures that `!=` is truly the inverse of `==`
-

Pop Quiz!

- You have two point objects

```
Point p1, p2;  
... // Initialize p1 and p2 coordinates, somehow.
```

- Now you want to scale them both by some factor

- ...and you decide to be clever...

```
Point &p = p1; // Make reference to p1  
p.setX(factor * p.getX()); // Scale p1  
p.setY(factor * p.getY());
```

```
p = p2; // Set reference to p2  
p.setX(factor * p.getX()); // Scale p2  
p.setY(factor * p.getY());
```

- Does this code work as intended?
 - If not, what does it actually do?

Assignment and C++ References

- Code:

```
Point &p = p1;           // Get a reference to p1
p.setX(factor * p.getX()); // Scale it
p.setY(factor * p.getY());
```

```
p = p2;                 // Get a reference to p2
p.setX(factor * p.getX()); // Scale it
p.setY(factor * p.getY());
```

- How does compiler interpret `p = p2;` line?

- ❑ `p.operator=(p2);`
 - ❑ Since `p` is a reference to the object `p1`, we overwrite `p1` with `p2`'s contents!
 - ❑ Original contents of `p1` are obliterated. `p2` isn't changed.
-

Moral: Know Your C++ References!

- C++ references work very differently from references in other languages
- Once you initialize a reference to refer to one object, you can't change it to refer to a different object

```
Point p1, p2;
```

```
...
```

```
Point &p = p1;
```

- `p` is a reference to `p1` from this point forward, until `p` goes out of scope

```
p = p2;
```

- This line means `p.operator=(p2)`, and is called on object `p1`
 - A reference is like an alias for another variable
-

This Week's Assignment

- Update your **Matrix** class to use **const**
 - Implement operator overloads for your class
 - Provide an assignment operator
 - **add()** and **subtract()** will go away, but you can use this code for the operators!
 - Implement matrix multiplication
 - Equality/inequality tests
 - Test code is provided!
 - Checks both functionality and **const** correctness
-

Homework Tips

- Matrix multiplication is tricky
 - Matrices don't have to be square
 - $[i \times j] [j \times k] \rightarrow [i \times k]$
 - `*=` operator can change dimensions of LHS
 - Tips for implementing `*=` operator:
 - Use a Matrix local variable to hold results of matrix-multiply
 - `Matrix tempResult(rows, rhs.cols);`
 - Perform multiplication into that local variable
 - `tempResult.setelem(r, c, val);`
 - Assign the local variable to `*this`
 - `*this = tempResult; // Assign result to myself`
 - Reuse your hard work on assignment operator! 😊
-

Other Tips

- Follow all the operator overload guidelines
 - Details are posted with homework
 - Test code will enforce some of these things too
 - Reuse your work
 - Have destructor and assignment operator use a helper function to do clean-up work
 - Have copy-constructor and assignment operator use another helper function to do copying
 - Implement `+`, `-`, `*` in terms of `+=`, `-=`, and `*=`
 - Implement `!=` in terms of `==`
-