CS11 Advanced C++
FALL 2015-2016
LECTURE 4
Unsigned Integers

- Unsigned integer values appear in numerous places, often as `size_t`
  - An unsigned integer value, often 64 bits these days
  - Typically defined in `<cstdlib>`
- Allocation functions like `malloc()` take allocation size as `size_t`
- STL collections return number of elements from `size()` as `size_t`
- `size_t` appears in many other places as well
- `for (int i = 0; i < vec.size(); i++) {...}
  - Compiler reports a warning: signed/unsigned comparison
Unsigned Integers (2)

- Typically just an annoyance
  ```cpp
  for (int i = 0; i < (int) vec.size(); i++) ...
  ```
- Can also lead to broken code!
- Example: iterate over a vector in reverse order, from end to start
  ```cpp
  for (unsigned int i = vec.size() - 1; i >= 0; i--)
  ```
- Problems?
  - This loop will never terminate!
  - When i is 0, i-- will wrap around to $2^{32} - 1$
  - i can never be negative! i will always be $>= 0$. 
In retrospect, the previous issue was pretty obvious.

Another example:
```c
int main() {
    long a = -1;
    unsigned b = 1;
    if (a < b)
        cout << "a < b" << endl;
    else
        cout << "a >= b" << endl;
    return 0;
}
```

What does this program output?
This program's output depends on whether it is compiled on a 32-bit platform or a 64-bit platform.

```c++
int main() {
    long a = -1;
    unsigned b = 1;
    if (a < b)
        cout << "a < b" << endl;
    else
        cout << "a >= b" << endl;
    return 0;
}
```

- On 64-bit, outputs “a < b” ✓
- On 32-bit, outputs “a >= b” ✗
To understand the program’s behavior, must understand C/C++ rules for integer conversions.

For all binary arithmetic / comparison operators, C/C++ follows specific rules to make the types of both sides compatible with each other.

Goals: avoid truncating values, and if possible, preserve the meanings of values.

Example:

```c
char c1(100), c2(3), c3(4), cr;
cr = c1 * c2 / c3;
```

What is `cr` at the end of the computation?
What is `cr` at the end of the computation?

```c
char c1(100), c2(3), c3(4), cr;
cr = c1 * c2 / c3;
```

C/C++ promote types smaller than `int` to `signed int` before performing arithmetic.

- `c1 * c2` is performed as `int` arithmetic, so 300 doesn't overflow the `char` type.

`cr` will be 75, as it should be 😊.
Integer Conversion Rules (3)

- For operations involving two integer values:
- If one value is signed and the other is unsigned:
- If all possible values of the unsigned type can be represented in the signed type, then the unsigned type is converted into the signed type.

Example:

```c
long long int a = -1;  // 64-bit signed int
unsigned int b = 1;    // 32-bit unsigned int
if (a < b)  // b converted to 64-bit signed int
    ...
```

- In this case, `a < b` will evaluate to `true`
Integer Conversion Rules (4)

- If the unsigned type is the same size as, or larger than, the signed type, then the signed type is converted into the unsigned type.

- Previous example, types revised:
  ```
  int a = -1; // 32-bit signed int
  unsigned int b = 1; // 32-bit unsigned int
  if (a < b) // a converted to 32-bit unsigned int
    ...
  ```

- Problem:
  - Signed 32-bit value of -1 == 4,294,967,295 as an unsigned 32-bit value
  - \( a < b \) will evaluate to \textit{false} 😞
Our Example Program

- Our program:
  ```c
  int main() {
    long a = -1;
    unsigned b = 1;
    if (a < b)
        cout << "a < b" << endl;
    else
        cout << "a >= b" << endl;
    return 0;
  }
  ```

- On 64-bit, signed long is 64-bit, unsigned int is 32-bit; converts b into 64-bit signed long

- Performs signed comparison between two 64-bit signed values – gives correct answer
Our Example Program (2)

- Our program:
  ```
  int main() {
    long a = -1;
    unsigned b = 1;
    if (a < b) {
      cout << "a < b" << endl;
    } else {
      cout << "a >= b" << endl;
    }
    return 0;
  }
  ```

- On 32-bit, signed `long` is 32-bit, unsigned `int` is 32-bit; converts `a` into 32-bit unsigned `int`

- Performs unsigned comparison between two 32-bit unsigned values – gives wrong answer 😞
Signed/Unsigned Conversion Bugs

- Compiler warnings are a good enough reason to fix this kind of thing...
- **Can also lead to security vulnerabilities in programs!**
- April 2008 – Adobe Flash Player vulnerability allows arbitrary code execution
- **Cause:** a signed/unsigned conversion issue
  - Flash Player passed a signed integer to `calloc()`, which takes an (unsigned) `size_t`
  - An attacker could affect this value, causing a negative number to be passed to `calloc()`
  - Signed value implicitly converted to unsigned value, which overflows, causing `calloc()` to return `NULL`
  - This opened the door to the vulnerability...
Friends

- Typically, access to class members is controlled by access modifiers
  - public – accessible to everyone
  - protected – accessible within declaring class and its subclasses
  - private – only accessible within declaring class

- Classes can also declare other classes and functions to be friends
  - The friend class / friend function is allowed to access the private members of the class

- Rationale: declaring other code as a friend can help preserve a class’ encapsulation
  - Class won’t have to expose as much on its public API
Friend Functions

Example:

class MyClass {
    friend void doStuff(const MyClass &m);
    int n;

public:
    ...
};

void doStuff(const MyClass &m) {
    cout << m.n << endl;
}

- doStuff() can access MyClass' private members
Friend Classes

Example:

```cpp
class C1 {
    friend class C2;
    int n;
    ...
};

class C2 {
    int m;
    public:
        C2(const C1 &c1) { m = c1.n; }
};
```

- C2 can access C1's private members
- **Note:** C1 cannot access C2's private members!
Friend Member-Functions

- Can even declare specific member functions as friends of a class

- Caveat: entire declaration of class with the friend member-function must precede the class that declares it as a friend
  - Otherwise, compiler won’t be able to verify the member function’s signature
Friend Member-Functions (2)

class C1;  // Forward declaration of class C1

class C2 {   // Declaration of class C2
public:
    void foo(const C1 &c);
};

class C1 {   // Declaration of class C1
    friend void C2::foo(const C1 &);
    ...
};

// Definition of C2::foo(), now that both C1 and C2 have been fully declared.
void C2::foo(const C1 &c) { ... }
C++ Nested Classes

- Can declare a class or struct inside of another class
  - Called a **nested class**
- Nested classes are automatically friends of their enclosing class
  - Can access all private/protected members declared in the enclosing class
- Important Caveat:
  - An object of the enclosing class type must be passed to the nested class for it to access the enclosing class’ members
  - (C++ nested classes are not like Java inner classes!)
Example 1:

```cpp
class Outer {
    int a, b; // Private members
    void foo();

    class Inner {
        void bar() {
            a += 5; // Compile error
            foo();  // Compile error
        }
    }
};
...
Example 2:
```cpp
class Outer {
    int a, b;  // Private members
    void foo();

    class Inner {
        // Reference to object of type Outer
        const Outer &outer;
        Inner(const Outer &o) : outer(o) { }

        void bar() {
            outer.a += 5; // OK
            outer.foo();  // OK
        }
    }
};
...
This Week’s Assignment

- Last assignment involving vector template (yay!)
- Create a template specialization for vectors of bools
  - Don’t want to use 32 bits to store each true/false value
  - Instead, store each Boolean value in a separate bit
  - a.k.a. a bit-vector
- C++11 has a new C-standard header <cstdint>
  - Defines integer types like int8_t, int32_t, int64_t
  - Unsigned integer types like uint8_t, uint32_t, uint64_t
  - Other interesting types like intptr_t – an integer type capable of holding a pointer
Vectors of bools

- Template specialization should use `uint32_t` type to store a vector of bits
  - Managing the vector’s capacity will change somewhat
  - You should be able to leverage your vector base-type from last assignment!
- Biggest challenge: accessing bit-vector elements
- Supporting rvalue access shouldn’t be hard:
  - Given the index of a bit to retrieve, can find the `uint32_t` value that will hold the bit
  - Can use bit-shift operations to read the specific bit
Vectors of bools (2)

- Lvalue access and iterators will be more interesting
  - ...how to create a reference to a specific bit within an array of uint32_t values?!?

- Example:
  ```cpp
  Vector<bool> bitVector(1000);
  bitVector[319] = somePredicate(x);
  ```
  - How to support writing to a specific bit?
Vectors of bools (3)

- Solution turns out to be pretty simple:
  - Create a helper class that handles assigning to a specific bit within the bit-vector
    - Take a reference to bit-vector, as well as index of bit to write to
    - Implement assignment operator, taking a bool as an argument – store the bool into appropriate bit in the bit-vector
  - An object of this type can be returned by Vector's non-const operator[](int index) impl.
    - The object is used as the target of the assignment
    - The object turns around and mutates the correct bit
The class that handles assignment must access the internals of your bit-vector

- Will need to either be a friend class, or a nested class
- (May be easier to use a nested class, since you are writing a template...)

Vectors of bools (4)
Iterators are only slightly more challenging

Another helper class that “acts like” a pointer into the bit-vector

Internally, will store a reference to the bit-vector, and the index of the bit referenced by the iterator

Iterator type needs to support increment/decrement and dereference

Increment/decrement operators can simply ++/-- the index that the iterator points to

Dereference operator can return the assignment-helper class to allow a single bit to be assigned to

For efficiency, should also provide a \texttt{const} version that simply returns a \texttt{bool} rvalue