CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Spring 2010
Lecture 23 - C/C++: const keyword
Software Eng: Defensive Programming
Where We Are

- One goal of the class is to help you become more mature software developers

- Last time, we talked about
  - Software development process
  - Writing specifications
  - Testing

- Today we will talk about the implementation
Good Software Development Practices

- **Writing readable code**
  - It's not just about the comments
  - Want whole program logic to be easy to follow

- **Writing code that is easy to maintain**
  - Well-defined components with clear interfaces
  - Loose coupling between components

- **Writing robust code that**
  - Gracefully reacts to unforeseeable usage
  - Gracefully handles various error conditions
  - **Software engineering principle: encapsulation**
What does the following code snippet do?

```c
int main(int argc, char** argv) {
    // ...
    int i[argc-1];
    for (int j=0; j<argc-1; i[j]=atoi(argv[++j]));
    cout << ((argc-1) % 2 ? 'y' : 'n') << endl;
    // ...
}
```
int main(int argc, char** argv) {
    // ...
    int size = argc - 1;
    int numbers[size];
    for ( int i = 0; i < size; i++ ) {
        numbers[i] = atoi(argv[i+1]);
    }

    if ( (size % 2) == 0 ) {
        cout << "Number of elements is even" << endl;
    } else {
        cout << "Number of elements is odd" << endl;
    }
    // ...
}
Why Is Readability Important?

- Your code is part of your documentation
  - Others need to understand it
  - You need to understand it, even after a while

- Maintenance
  - Fixing bugs is easier when code is readable
  - Adding new features is also easier

- Clear code helps clear thinking

- If your code is unreadable, it will quickly end-up in the garbage
How to Improve Code Readability

- It's not just about the comments
- **Use good levels of abstraction**
  - Each function should have a single specific goal
  - The algorithm used by the function should be clear
  - Use small helper functions to hide details
- **Make program logic easy to follow**
- **Some small things that also help**
  - Write clear expressions and statements
  - Good variable names and indentation
  - Follow a coding standard
Good Software Development Practices

- **Writing readable code**
  - It's not just about the comments
  - Want whole program logic to be easy to follow
- **Writing code that is easy to maintain**
  - Well-defined components with clear interfaces
  - Loose coupling between components
- **Writing robust code that**
  - Gracefully reacts to unforeseeable usage
  - Gracefully handles various error conditions

- **Software engineering principle: encapsulation**
Writing Robust Code

- Defensive programming
  - Check your function inputs
  - Check buffer boundaries
  - Check for errors, catch and handle exceptions
- Enforce encapsulation (data hiding)
  - Important software engineering principle
- Other general practices
  - Strive for simplicity, perform code reviews
  - Check invariants (helps testing/debugging)
    - Example: list is always in sorted order
  - Reuse well-tested code: standard libraries
Check Your Function Inputs

- Famous last words:
  - “No one will pass null as argument. Why would they?”
  - “No one will ever enter a name longer than X”
  - “I will first get it to work. I will add all the error handling later, when I have time”

- Golden rules
  - Assume callers do not know what they are doing
  - Check that inputs are valid!
  - Check preconditions if possible
Check Your Function Inputs

- Example from `StringList.cc`
- Always check inputs! Handle errors as per specs
- Check preconditions if possible
  - For preconditions, `assert` is very convenient

```cpp
void StringList::insert (const char *original) {
  // CHECK: Checking all inputs
  // CHECK: Checking preconditions
  assert( original );
  assert( strlen(original) < BUF_SIZE );
  ...
}
```
Check Buffer Boundaries

- Every time you manipulate an array or string
  - Make sure you are staying within bounds

- Example from StringList.cc

```cpp
void StringList::insert(const char* original) {
    Node node = new Node();
    ...
    strncpy(node->original, original, BUF_SIZE);
    ...
}
```
Every time you invoke a function

- Check if the function can return an error
  - Read the specification for that function
  - One reason why good specifications are important
- Assume it will sometimes return that error
- Handle the error properly

Many examples

- Opening a file can fail (fopen)
- Reading data from a stream can fail (fscanf)
- Etc.
Check For Errors

- Example from `StringList.cc`

```c++
void StringList::insert (const char *original) {
    Node *node = new Node();
    if ( !node ) {
        cerr << "Out of memory\n";
        return;
    }
    ...
}
```
Writing Robust Code

- Defensive programming
  - Check your function inputs
  - Check buffer boundaries
  - Check for errors, catch and handle exceptions
- Enforce encapsulation (data hiding)
  - Important software engineering principle
- Other general practices
  - Strive for simplicity, perform code reviews
  - Check invariants (helps testing/debugging)
    - Example: list is always in sorted order
  - Reuse well-tested code: standard libraries
Encapsulation

- Key concept in object-oriented programming
- A class encapsulates attributes and functions
  - Classes correspond to “abstract data types”
  - A class “exports” an interface
  - All communication goes through interface
  - No one is allowed to manipulate data directly
- Information hiding
  - No one should know about implementation nor representation (the internal data structures of the class)
- Example: StringList class
  - User of the class does not know how list is implemented
Check Invariants

- Internal class representation often has some invariants: i.e., properties that always hold
- Example of invariant:
  - “Linked list is always in sorted order”
- **Add a function**: check_list
  - Returns `true` if function is in order
  - Returns `false` otherwise
- **Inside your functions**: insert and delete
  - Add: `assert(check_list(*head));`
- This practice helps early bug detection
Information Hiding Common Error

- It is easy to break encapsulation by accident
- Typical problem: caller and callee have pointers to the same object
- Caller can change internal representation of the callee! Very BAD!
- A very common source of errors
Information Hiding Common Error

- Example 1: Error when handling inputs

```c
void StringList::insert(const char *original) {
    Node *node = new Node();
    ...
    node->original = original;
}
```

- In the example above, the caller and callee point to the same array of characters in memory. This is bad.
Information Hiding Common Error

- Example 2: Error when handling outputs

```cpp
Const Node*
StringList::lookup (const char *original) {

    Node *element = _head;
    // Iterate through list and find string
    // ...
    return element;
}
```

- In this example, the caller and callee point to the same Node element in memory. This is bad even with a const qualifier
Information Hiding Common Error

- In the lookup example, caller cannot change the element returned: **GOOD**
- However, caller can still free the object: **BAD**
- Also, caller has a pointer to an element that someone else can free by removing the string from the list: **BAD**
Information Hiding Solutions

- **Solution 1:** Copying
  - Copy all inputs before integrating them into internal representation
  - Return copies of internal elements

- **Solution 2:** Immutable objects
  - Immutable objects can never be changed
  - But watch-out for new/delete

- **Solution 3:** Using the `const` type qualifier
  - Good idea, but be careful
  - Once again, watch-out for new/delete
The “const” Type Qualifier

- Available in C and in C++
- Enforced at compile time

Example 1: Using const with inputs

```c
void StringList::insert(const char *original) {
    // Following causes compile-time error
    original[0] = ...;
}
```
The “const” Type Qualifier

- Example 2: Using const with return values

```c++
const Node*
StringList::lookup (const char *original) {

    Node *element = head;
    // ...
    return element;
}
// Caller cannot change the element returned
const Node *element = list.lookup(my_string);
// And following causes compile time error
element->original[0] = 'a';

- BUT, caller can still delete object, so we would still want to make a copy instead of returning a const pointer
```
“const” Can Get Very Confusing

- Non-constant pointer to constant data
  - const char *ptr
  - Cannot change the content of these locations
  - Can make ptr point to different memory locations
- Constant pointer to non-constant data
  - char * const ptr = ...;
  - Cannot change what ptr is pointing to
  - Can change the content of pointer to location
- Can also have **const pointer to const data** and a **non-const pointer to non-const data**
Basic Principle

- **Principle of least privilege**
  - Give a function enough access to data to accomplish task. Not more.

- **Note: in C++, you can declare a member function inside a class to be const**
  - Means that function is not allowed to modify any data members
  - Simply specify keyword const at end of prototype

```c
void print() const;
bool is_empty() const;
```
Writing Robust Code

- Defensive programming
  - Check your function inputs
  - Check buffer boundaries
  - Check for errors, catch and handle exceptions
- Enforce encapsulation (data hiding)
  - Important software engineering principle
- Other general practices
  - Strive for simplicity, perform code reviews
  - Check invariants (helps testing/debugging)
    - Example: list is always in sorted order
  - Reuse well-tested code: standard libraries
Towards Security

- Robust software can protect against
  - Buffer overflow attacks
  - Crashes caused by invalid inputs

- But security is much harder than that

- Example 1: denial of service attack
  - Send huge numbers of requests to a server
  - For example, keep adding elements to list

- Example 2: timing attack
  - Measure time system takes to fulfill a request
  - Example: timing.c
You now know some **basic software engineering**

- **Software development process**
  - Main steps involved in building a software system

- **Specifications**
  - Why we need them and how to write simple ones
  - We talked about informal specifications only

- **Testing:** why and how

- **Writing robust and readable code**

There is much more to software engineering

But what you know should help in future classes