Software Engineering Concepts: Testing
Integer Types
Managing Resources in a Class

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 2, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- Integer types
- Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

Major Topics: S.E. Concepts

- ✓ Invariants
- Testing
- Abstraction

The "Simple Class Example" is finished. See `timeofday.h` & `timeofday.cpp`.

Review

Review
Software Engineering Concepts: Invariants

An **invariant** is a condition that is always true at a particular point in an algorithm.

Three Special Kinds

- **Precondition**. An invariant at the beginning of a function. The responsibility for making sure the preconditions are true rests with the calling code.
  - What must be true for the function to execute properly.
- **Postcondition**. An invariant at the end of a function. Tells what services the function has performed for the caller.
  - Describe the function's effect using statements about objects & values.
  - Pre- and postconditions are the basis for **operation contracts**.
- **Class invariant**. An invariant that holds whenever an object of the class exists, and execution is not in the middle of a public member function call.
  - Statements about data members that indicate what it means for an object to be valid or usable.

Software Engineering Concepts: Invariants — EXERCISE

Exercise

- Write pre- and postconditions for the one-parameter constructor for class `Abc`.

Answers

See next slide.

```
// class Abc
// Invariants:
//     0 <= _n && _n < 100
class Abc {
public:
    Abc(int nn)
        :_n(nn)
    {}
    [other stuff here]
private:
    int _n;
};  // End class Abc
```

Exercise

- Write pre- and postconditions for the one-parameter constructor for class `Abc`.

```
// class Abc
// Invariants:
//     0 <= _n && _n < 100
class Abc {
public:
    Abc(int nn)
        :_n(nn)
    {}
    [other stuff here]
private:
    int _n;
};  // End class Abc
```

Answers

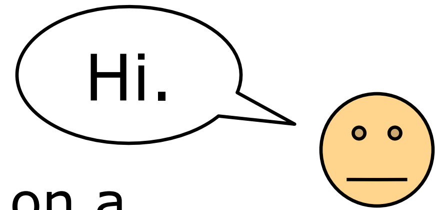
```
// Pre:
//     0 <= nn && nn < 100
// Post:
//     _n == nn
```

`0 <= _n && _n < 100`
is also a postcondition. But that is already in the class invariants, and we do not need to repeat it.

Software Engineering Concepts: Testing

Software Engineering Concepts: Testing
A Tragic Story [1/4]

Meet Egbert.



Egbert is a software developer. He is working on a project for a customer. It requires him to write three functions.

```
double mu(int n);     // Returns nasal perspicacity of n
void mumu(int n);     // Like mu, only different
int mumumu(int n);    // Like mumu, only more different
```

Egbert writes function `mu`. When he finishes, he starts on `mumu`, little knowing that he is ***making a terrible mistake!***

Cue ominous music …

Software Engineering Concepts: Testing
A Tragic Story [2/4]

… after a great effort, the deadline arrives. But Egbert is not done. However, he does have some code written. Here is what he has.

```
double mu(int n)  // Returns nasal perspicacity of n
{
    [amazingly clever code here]
}

void mumu(int n)  // Like mu, only different
{
    [heart-breakingly brilliant code here]
}

// TO DO: write function mumumu
```

Software Engineering Concepts: Testing
A Tragic Story [3/4]

Egbert meets with the customer. He explains that he is not done. The customer is a bit annoyed, of course, but he knows that schedule overruns happen in every business.

So, he asks, “Well, what *have* you finished? What can it do?”

But one of Egbert’s functions does not exist at all. So his unfinished package, when combined with the code that is supposed to use it, *does not compile*, much less actually execute.

He tells the customer, “Well, it doesn’t *do* anything. But it’s beautiful! Want to see the code?”

“No,” replies the customer, through clenched teeth.

The customer storms off and screams at Egbert’s boss, who confronts Egbert and says he had better have something good in a week. Egbert gives his solemn assurance that this will happen.

He goes back to work …

Software Engineering Concepts: Testing
A Tragic Story [4/4]

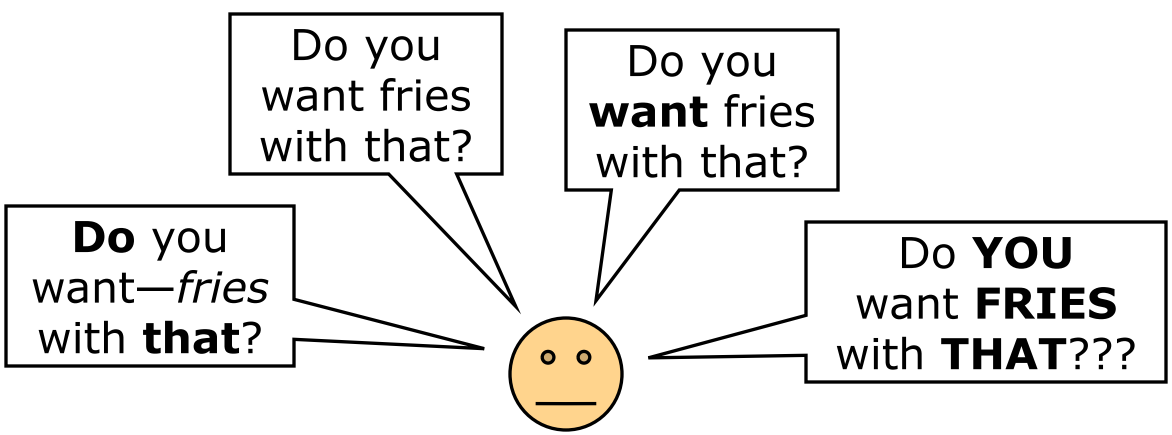
… and writes a do-nothing function `mumumu`, just to get things to compile.

However, when he does this, he realizes that, since he has never compiled the package, he has never tested *anything*—even the three functions he thinks of as “finished”.

Now that he *can* test them, he finds that they are full of bugs.

Alas, he now knows that he have been far too optimistic; nothing worthwhile is going to get written in the required week.

He begins practicing his lines for an exciting new career …



Revising the Process [1/4]

Observations

- Code that does not compile is worthless to a *customer*, even if it is "mostly finished".
- It might not be worth anything to *anyone*; **you cannot tell**, because …
- Code that does not compile cannot be tested, and so it might be much farther from being really *done* than you suspect.
- Testing is our primary method of uncovering bugs.

Conclusion

- First priority: **Get your code to compile**, so that it can be tested.

Revising the Process [2/4]

A Revised Development Process

- Step 1. Write dummy versions of all required components.
  - Make sure the code **compiles**.
- Step 2. Fix every bug you can find.
  - “Not having any code in the function body” is usually a bug.
  - Write notes to yourself in the code.
  - Make sure the code **works**.
  - In this step, the code should always compile.
- Step 3. Put the code into final, deliverable form.
  - The code needs to be pretty, well commented/documented, and in line with coding standards.
  - Many comments can be based on notes to yourself.
  - Make sure the code is **finished**.
  - In this step, the code should always work.

There is a lot more to say about software development processes. *See CS 372.*

Revising the Process [3/4]

Suppose Egbert had used this revised development process earlier.

Step 1. Write dummy versions of all required components.

```
double mu(int n)  // Returns nasal perspicacity of n
{ return 1.; }  // Dummy; TODO: write function body

void mumu(int n)   // Like mu, only different
{}               // TODO: write function body

int mumumu(int n)  // Like mumu, only more different
{ return 1; }   // Dummy; TODO: write function body
```

Does it compile? Yes. Step 1 is finished.

Software Engineering Concepts: Testing
Revising the Process [4/4]

Step 2. Fix every bug you can find.

Egbert begins testing the code. Obviously, it performs very poorly. But he begins writing and fixing. And running the code. So when something does not work, *he knows it*. When he figures something out, he makes a note to himself about it.

The deadline arrives, but the code is not finished yet.

Egbert meets with the customer. “The project is not finished,” he says, “but ***here is what it can do***.”

He estimates how long it will take to finish the code.

He can make this estimate with some confidence, because he has a list of tests that do not pass; he knows what needs to be done.

Software Engineering Concepts: Testing
Unit Testing

There are many kinds of software testing. One important kind is **unit testing**—tests for the various **units** in the code (functions, classes, etc.), individually.

Unit testing is common enough that high-quality unit-testing **frameworks** (also called **harnesses**, for some reason) are available for most/all major programming languages.

This semester, I will give you test programs for 7 of the 8 projects. These will do unit testing. My test programs use a C++ unit-testing framework called **doctest**.

Integer Types

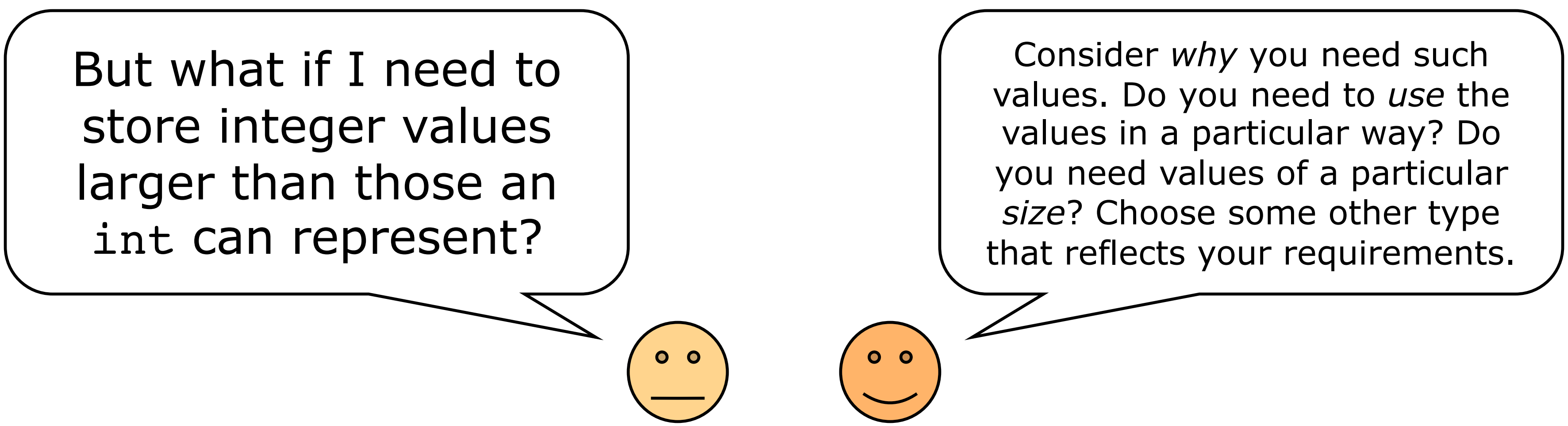
Integer Types
Built-In Integer Types

C++ includes a number of built-in **integer types**: `int`, `short`, `long`, `long long`. Also `unsigned int`, `unsigned short`, etc.

Which to use?

- If you call someone else's code, or write based on someone else's specification, then you use whatever types you must.
- But when you get to choose the types …

Suggestion. Out of the above, directly use only `int`.



Next we look at some other integer types you might use.

Integer Types
Standard Library Integer Types [1/2]

The C++ Standard Library defines integer types that reflect certain *intended uses*. Examples:

- Type `std::size_t` The ending "`t`" stands for "type".
  - An unsigned (cannot be negative) integer type big enough to hold the size of any object in memory.
  - Declared in header `<cstddef>`.
- Type `std::ptrdiff_t`
  - Much like size_t, but signed (can be negative). Gets its name from the fact that it can hold the result of subtracting two pointers.
  - Declared in header `<cstddef>`.

`std::size_t` is *probably* an alias for `unsigned long`. However, `size_t` is better than `unsigned long`, because:

- It works on all systems, for holding sizes of in-memory objects.
- It gives the reader an idea what values are used for.

`std::size_t` is a good choice for container sizes and indices.

Integer Types
Standard Library Integer Types [2/2]

The C++ Standard Library also defines integer types that reflect particular *storage sizes*. Examples:

- Type `std::int64_t`
  - A signed integer type taking up exactly 64 bits.
  - Declared in header `<cstdint>`.
- Type `std::uint64_t`
  - An unsigned integer type taking up exactly 64 bits.
  - Declared in header `<cstdint>`.
- Type `std::uint_fast64_t`
  - An unsigned integer type taking up at least 64 bits, and, out of all such types, having the fastest operations.
  - Declared in header `<cstdint>`.

By the way, how do I know these are declared in `<cstdint>`?

And how do I know exactly where the underscores go?

Integer Types
Helpful Member Types

Classes may have **member types** that are similarly useful.

- For example, `vector` has member types `size_type`, `value_type`.

```
vector<Foo> v;
auto howbig = v.size();  // Type: vector<Foo>::size_type
```

We can make our own member types.

```
class FooList {
public:
    using size_type = size_t;
    using value_type = Foo;
```

Member types

Client code can now use `FooList::size_type` and `FooList::value_type`.

Managing Resources in a Class

Recall that a **pointer** holds the **address** of another value.

```
int n;
int * p = &n;
```

`p` is a pointer.
"`&`" is the **address-of operator**.

Now `*p` is the same as `n`.

"`*`" is the **dereference operator**.

```
p = nullptr;
```

A **null pointer** does not point at anything.

Also, we can check for it: "`if (p == nullptr)`".

```
p = new int;
…
delete p;
```

Dynamic allocation

Deallocation

Managing Resources in a Class
Preliminaries — Pointers [2/2]

For each `new`, there must be a `delete`; otherwise, there is a **memory leak**.

Q. What does the destructor of a pointer do?

A. ???

What happens here???

```
void gg()
{
    auto p = new int;
    *p = 42;
}
```

Managing Resources in a Class
Preliminaries — Pointers [2/2]

For each `new`, there must be a `delete`; otherwise, there is a **memory leak**.

Q. What does the destructor of a pointer do?

A. *Nothing!*

Nothing happens here.

```
void gg()
{
    auto p = new int;
    *p = 42;
}
```

Managing Resources in a Class
Preliminaries — Exceptions

When a function encounters an *error condition*, this often needs to be communicated to the caller (or the caller's caller, or the caller's caller's caller, …).

One way to do this is by **throwing an exception**.

- This causes control to pass to the appropriate handler, which **catches** the exception.
- When an exception is thrown, a function can exit in the middle, despite the lack of a `return` statement.

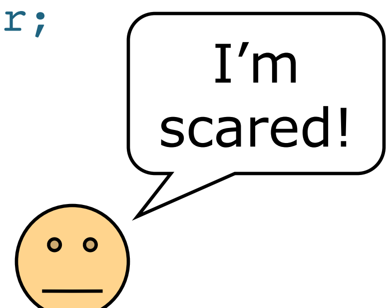
We discuss exceptions in a few days. For now, be aware that:

- Exceptions can result in a function exiting just about anywhere.
  - In particular, if function `foo` calls function `bar`, and function `bar` throws, then function `foo` will exit if it does not catch the exception.
- When a function exits, whether by a normal `return` or by throwing an exception, destructors of all automatic objects are called.

Managing Resources in a Class
Problem & Solution — The Problem

What is *scary* about code like this?

```
void scaryFn(size_t size)
{
    int * buffer = new int[size];
    if (func1(buffer))
    {
        delete [] buffer;
        return;
    }
    if (func2(buffer))
    {
        delete [] buffer;
        return;
    }
    func3(buffer);
    delete [] buffer;
}
```



Function `scaryFn` has 3 exit points.

- The buffer must be freed in each.
- Otherwise, it will never be freed. This would be a memory leak.

If we alter the code in this function, it is easy to create a memory leak accidentally.

In fact, there may be other exit points, if one of the 3 functions called ever throws an exception.

- In that case, function `scaryFn` has a memory leak already.

Now imagine a different scenario: some memory is allocated and freed in different functions.

- What if it might be freed in *one of several* different functions?
- Memory leaks become hard to avoid.

Managing Resources in a Class
Problem & Solution — About Destructors

We want to solve this problem.

Recall the rules for when destructors are executed:

- The destructor of an **automatic** (local non-static) object is called when it goes out of scope.
  - This is true no matter whether the block of code is exited via `return` (functions), `break` (for loops), `goto` (ick!), hitting the end of the block of code, or an exception.
- The destructor of a **static** (global, static local, or static member) object is called when the program ends.
- The destructor of a non-static **member** object is called when the object of which it is a member is destroyed.

So we can depend on execution of destructors, except for:

- **Dynamic** objects (those created with `new`).

Therefore …

Managing Resources in a Class
Problem & Solution — A Solution: RAII

Solution

- Each dynamic object, or block of dynamically allocated memory, is managed by some other object.
- In the **destructor** of the managing object:
  - The dynamic object is destroyed.
  - The dynamically allocated memory is freed.

Results

- Destructors always get called.
- Dynamically allocated memory is always freed.

This programming idiom is, misleadingly, called **Resource Acquisition Is Initialization** (**RAII**).

- The name would seem to mean that allocation is done in the various constructors. In practice, we might do that, or we might not.
- But we always **deallocate in the destructor**—if the memory in question has not been deallocated by that point.

Managing Resources in a Class
Problem & Solution — Ownership

In general (RAII or not), to avoid memory leaks, we need to be careful about which component is responsible for freeing a block of memory or destroying a dynamic object.

Whatever has this responsibility is said to **own** the memory/object.

For example, a function can own memory.

- This is what we saw in function `scaryFn`.

RAII means that a dynamic object or block of memory is owned by some other *object*.

Ownership = Responsibility
for Releasing

RAII = An Object Owns
(and, therefore, its destructor releases)

Managing Resources in a Class
An RAII Class — Getting Started

Rather than have an object *directly* manage every resource it deals with, we can use wrapper classes that do RAII.

Let's write a simple RAII class that owns a dynamic integer array.

- Call it `IntArray`.
- What is the **absolute minimum functionality** that such a class must have, to be useful in improving a function like `scaryFn`?
  - Creation (ctor from size?)
  - Destruction
  - Item access (bracket op?)
- Rewrite `scaryFn` to use this new class.

```
void scaryFn(size_t size)
{
    int * buffer = new int[size];
    if (func1(buffer))
    {
        delete [] buffer;
        return;
    }
    if (func2(buffer))
    {
        delete [] buffer;
        return;
    }
    func3(buffer);
    delete [] buffer;
}
```

Managing Resources in a Class
An RAII Class — Constness

We want to be able to change items in a normal `IntArray`, but not in a `const IntArray`.

```
IntArray nc(20);                      const IntArray c(20);
cout << nc[1];  // Legal              cout << c[1];  // Legal
nc[1] = 2;      // Legal              c[1] = 2;      // NO!
```

Q. How can we make `IntArray` work this way?

A. Have two versions of the bracket operator, one non-const, one const. They are identical, except for the types involved.

This idea is common, when dealing with access to data managed by an object.

```
int & operator[](size_type index)
{ return arrayPtr_[index]; }
const int & operator[](size_type index) const
{ return arrayPtr_[index]; }
```

Managing Resources in a Class
An RAII Class — Keyword explicit

Implicit type conversion: invisible function call, converts types. Some are built-in, like the implicit conversion from `int` to `double`.

```
void foo(double x);
foo(3);       // 3: int, not double; implicit conversion
```

A one-parameter ctor can be used to do implicit type conversions *unless* it is declared **explicit**.

```
class IntArray {
public:
    explicit IntArray(size_type size);
```

What type conversion would this ctor do, if it were not explicit?

We often declare one-parameter ctors explicit.

- Not copy/move ctors! That would disallow passing by value.

TO DO

- Write class `IntArray`.
  - Constructor from size (explicit).
  - Destructor.
  - Bracket operator (both const & non-const).
  - Member types `size_type`, `value_type`.
- Rewrite function `scaryFn` to use `IntArray`.

Partially done. See `intarray.h`*.*
See `intarray_main.cpp`
for a simple main program.

Next time.

Managing Resources in a Class
TO BE CONTINUED …

Managing Resources in a Class will be continued next time.