Casting & std::optional

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat

Mati Davis

Chandravaran Kunjeti

Shufan Liu

Jialin Cai

Donglun He

Heyi Liu

Eddy Yang

Poll Everywhere

pollev.com/tqm

- What courses are you planning to take next?

Poll Everywhere

pollev.com/tqm

- Any questions from previous lectures?

Logistics

- HW4 Posted Due Thursday 4/20 @ 11:59

- Project Released! Due Wednesday 4/26 @ 11:59

- HW2 grades & Midterm grades posted later today
  - Can fix HW2 submissions
  - Midterm has regrades & the clobber policy

Logistics

- Final Exam Scheduling:
  - 96 hours (4 days)
  - Opens Tuesday May 2nd @ Noon
  - Closes Saturday May 6th @ noon

- Extra OH today & next Monday from Kevin @ Levine 501, 5-7 pm

Lecture Outline

- **C++ Inheritance**
  - Static Dispatch
  - **Constructors and Destructors**
  - **Assignment**
- "Modern C++"
  - C++ Casting
  - std::optional & others

- Reference: *C++ Primer*, Chapter 15

Constructors and Inheritance

- A derived class **does not inherit** the base class' constructor
  - The derived class must have its own constructor
  - A synthesized default constructor for the derived class first invokes the default constructor of the base class and then initialize the derived class' member variables
    - Compiler error if the base class has no default constructor
  - The base class constructor is invoked *before* the constructor of the derived class
    - You can use the initialization list of the derived class to specify which base class constructor to use

Constructor Examples

badctor.cc

```
class Base {  // no default ctor
 public:
  Base(int yi) : y(yi) { }
  int y;
};

// Compiler error when you try to
// instantiate a Der1, as the
// synthesized default ctor needs
// to invoke Base's default ctor.
~~class Der1 : public Base {~~
~~ public:~~
~~  int z;~~
~~};~~

class Der2 : public Base {
 public:
  Der2(int yi, int zi)
    : Base(yi), z(zi) { }
  int z;
};
```

Compiler error ☹ No default ctor

Invokes a specific ctor

goodctor.cc

```
// has default ctor
class Base {
 public:
  int y;
};

// works now
class Der1 : public Base {
 public:
  int z;
};

// still works
class Der2 : public Base {
 public:
  Der2(int zi) : z(zi) { }
  int z;
};
```

Because base has default ctor

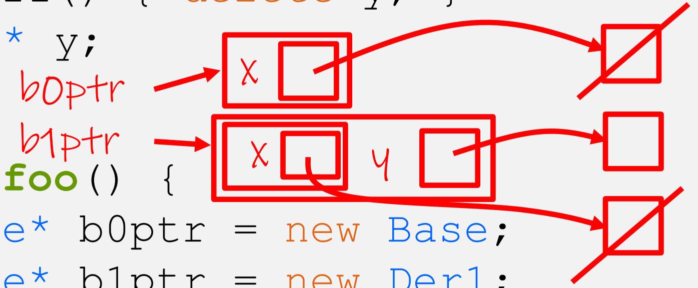
Destructors and Inheritance

- Destructor of a derived class:
  - *First* runs body of the dtor
  - *Then* invokes of the dtor of the base class

- Static dispatch of destructors is almost always a mistake!
  - Good habit to always define a dtor as virtual
    - Empty body if there's no work to do

baddtor.cc

```
class Base {
 public:
  Base() { x = new int; }
  ~Base() { delete x; } Not virtual,
                        Static dispatch
  int* x;
};

class Der1 : public Base {
 public:
  Der1() { y = new int; }
  ~Der1() { delete y; }
  int* y;
};

void foo() {
  Base* b0ptr = new Base;
  Base* b1ptr = new Der1;

  delete b0ptr;   // delete's x
  delete b1ptr;   // delete's x, but not y
}     Both invoke Base dtor!!!!
```

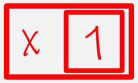


Assignment and Inheritance

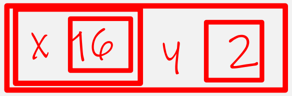
- C++ allows you to assign the value of a derived class to an instance of a base class
  - Known as object slicing
    - It's legal since `b = d` passes type checking rules
    - But `b` doesn't have space for any extra fields in `d`

slicing.cc

```
class Base {
 public:
  Base(int xi) : x(xi) { }
  int x;
};
```



```
class Der1 : public Base {
 public:
  Der1(int yi) : Base(16), y(yi) { }
  int y;
};
```



```
void foo() {
  Base b(1);
  Der1 d(2);

  d = b;   // Compiler error – not enough info
  b = d;   // ok, What happens to y?
           Y is not copied over.
}
```

STL and Inheritance

- Recall: STL containers store **copies of values**
  - What happens when we want to store mixes of object types in a single container? (*e.g.* `Stock` and `DividendStock`)
  - You get sliced ☹

```
#include <list>
#include "Stock.h"
#include "DividendStock.h"

int main(int argc, char** argv) {
  Stock s;
  DividendStock ds;
  list<Stock> li;

  li.push_back(s);    // OK
  li.push_back(ds);   // OUCH!

  return EXIT_SUCCESS;
}
```

STL and Inheritance

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! ☺ Vector<Stock*>
  - `sort()` does the wrong thing ☹ Sorts by address value on default
  - You have to remember to `delete` your objects before destroying the container ☹
    - Unless you use Smart pointers!

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Constructors and Destructors
  - Assignment
- **"Modern C++"**
  - **C++ Casting**
  - std::optional & others

- Reference: *C++ Primer*, Chapter 15

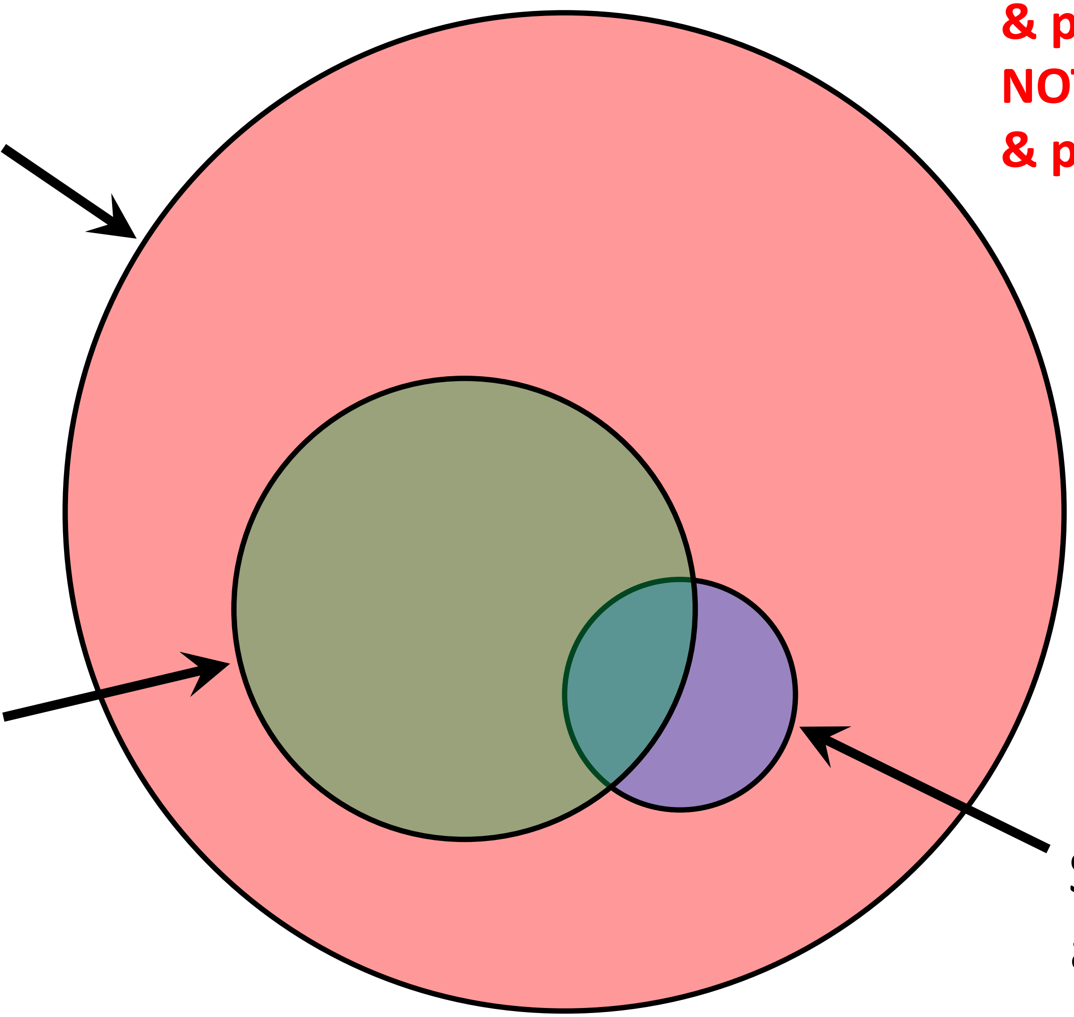
Modern C++ in this course?

- This course did not teach “the best” way to code in C++
  - This is a systems programming course, not a C++ course

- Many goals in this course:
  - Give you core systems knowledge
  - Prepare you for future courses
    - Some are in C
    - Some are in C++

- C is NOT C++ and vice-versa

Previously: How to Think About C++



Set of styles and ways to use C++

Good styles and robust engineering practices

Set of styles and ways to use C

What is good style & practice in C may NOT be good style & practice in C++

Modern C++: what is it?

- What is modern changes, but it is making use of the modern features of C++

- This includes
  - Vast use of C++ STL
    - vector, map, list, set, pair
  - Range for loops
    - E.g. **`for (auto& e : vec) {`**
  - Exceptions
  - RAII
  - …

Modern C++: what is it?

- This also means almost completely moving away from C style and doing things with C++
  - Stop using **`char*`**, use **`std::string`**
  - Stop using C style array, use **`std::array`**
  - Stop using C-style casts, uses C++ casts (more in a second)
  - Mostly avoid **`malloc()/free()`** and **`new/delete`**.
    - **`make_unique`** and **`make_shared`**
    - STL containers
  - Stop returning an int and using output params
    - Use structured binding, **`std::optional`**, **`std::variant`**, etc…

- Unavoidable at times, if the intention is to interface with C code ☹

Continuing to learn C++

- There is so much to C++, you do not have to know it all to be a good C++ dev.

  - **Practice makes perfect**
  - Before you can be kinda good at something, you have to be bad at it first

- Many resources out there, here is one:
  - C++ Core Guidelines
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
  - Goes over what is good practice and what is not. Not everything may make sense, that is ok. Take it slow, feel free to skip around

Explicit Casting in C

- Simple syntax:

```
lhs = (new_type) rhs;
```

- Used to:
  - Convert between pointers of arbitrary type (void*) my_ptr
    - Doesn't change the data, but treats it differently
  - Forcibly convert a primitive type to another (double) my_int
    - Actually changes the representation

- You *can* still use C-style casting in C++, but sometimes the intent is not clear

Casting in C++

- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

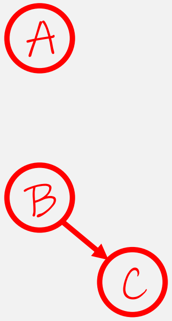
static_cast

Any well-defined conversion

- static_cast can convert:
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - casting void* to T*
  - Non-pointer conversion
    - *e.g.* float to int
- static_cast is checked at compile time

staticcast.cc

```
class A {
 public:
  int x;
};

class B {
 public:
  float y;
};

class C : public B {
 public:
  char z;
};
```



```
void foo() {
  B b; C c;

  // compiler error Unrelated types
  A* aptr = static_cast<A*>(&b);
  // OK  Would have worked without cast
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}   What happens when you do cptr->z?
```

dynamiccast.cc

dynamic_cast

- `dynamic_cast` can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- `dynamic_cast` is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type
- Can be used like `instanceof` from java

```
class Base {
 public:
  virtual void foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- const_cast adds or strips const-ness
  - Dangerous (!)

```
void foo(int* x) {
  *x++;
}

void bar(const int* x) {
  foo(x);                          // compiler error
  foo(const_cast<int*>(x));   // succeeds
}

int main(int argc, char** argv) {
  int x = 7;
  bar(&x);
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.* storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is “wide” enough
  - Converting between incompatible pointers
    - Dangerous (!)
  - Use any other C++ cast if you can.

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Constructors and Destructors
  - Assignment
- **"Modern C++"**
  - C++ Casting
  - **std::optional & others**

Functions that sometimes fail

- It is pretty common to write functions that sometimes fail. Sometimes they don't return what is expected

- Consider we were building up a Queue data structure that held strings, that could
  - Add elements to the end of a sequence

    -
      ```
      void add(string data);
      ```
  - Remove elements from the beginning of a sequence
    ```
    ???? remove(????);
    ```
  - How do we design this type to

Previous ways to handle failing functions

- Return an "invalid" value: e.g. if looking for an index, return -1 if it can’t be found.
  - What if there is no nice "invalid" state?

```
// what is an invalid string?
string remove();
```

- C-style: return an error code or success/failure.
  Real output returned through output param

```
bool remove(string* output);
```

Previous ways to handle failing functions

- Return a pointer to a heap allocated object, could return **`nullptr`** on error
  - Uses the heap when it is otherwise unnecessary ☹
  - Need to remember to **`delete`** the string

```
string* remove();
```

- Java style: throw an exception in the case of an error
  return the value as normal
  - Exceptions not best for performance
  - Exception catching not always the easiest to handle

```
string remove() {
  if (this->size() <= 0U) {
    throw std::out_of_range("Error!");
  }
```

std::optional

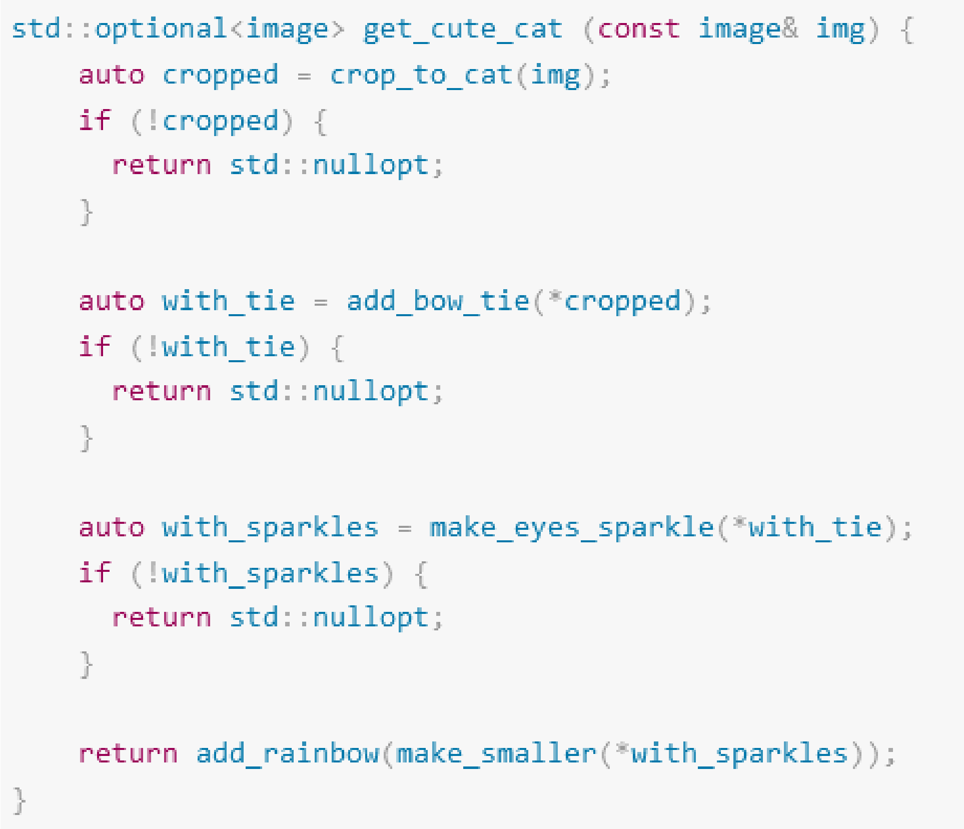
- **`optional<T>`** is a struct that can either:
  - Have some value T
    (`optional<string> {"Hello!"}`)
  - Have nothing
    (**`nullopt`**)

- **`optional<T>`** effectively extends the type **`T`** to have a "null" or "invalid" state

- How is this much better at all?
  - Code demo: Queue.h and use_queue.cc

Monadic optional

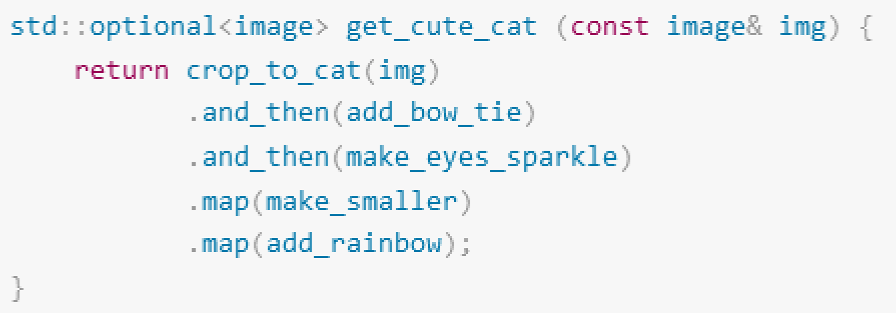
- If all we had from optional<T> was that it could be something or nothing, then our error handling code would still just be a bunch of if statements



```cpp
std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
      return std::nullopt;
    }

    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
      return std::nullopt;
    }

    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
      return std::nullopt;
    }

    return add_rainbow(make_smaller(*with_sparkles));
}
```

Monadic optional

- As of C++ 23, std::option can be used with new member functions



```
std::optional<image> get_cute_cat (const image& img) {
    return crop_to_cat(img)
           .and_then(add_bow_tie)
           .and_then(make_eyes_sparkle)
           .map(make_smaller)
           .map(add_rainbow);
}
```

- and_then
- map (now called transform)
  - These functions call the specified function on the value in the option, or just return **`nullopt`** if it is not available.

- See **`use_queue.cc`** for an example

Optional in other languages

- Languages which have their own optional-like type with this monadic interface:
  - Java
  - Swift
  - Haskell
  - Rust
  - Ocaml
  - Scala
  - Agda
  - Idris
  - Kotlin
  - StandardML
  - C#

Other ways to return: std::variant

- If your function could return **one of** two or more different values, could use std::variant, which indicates it could be any of the specified types

```
variant<int, float, string> get_some_value();
```

Other ways to return: Structured Binding

- If your function could return two or more different values at the same time could use a struct, tuple or pair

```
pair<int, string> get_some_value();
```

- Could access the values manually:

```
pair<int, string> p = get_some_value();
int x = p.first();
string y = p.second();
```

- Or use structured binding:

```
auto [x, y] = get_some_value();
// x and y both exist as variables
// that can be used!
```

C++23 and beyond!

- C++ is still being worked on, with many useful features!

- Don't like #include and dealing with weird header files?
  - C++ 20 added import statements, can write things like **`import std.regex`** and give more explicit control of what is visible to others
- Don't like how **`cout << "hello" << endl`**?
  - C++23 is adding std::print. E.g:
    - **`println("hello!");`**
    - **`print("{0} {2}{1}!\n", "Hello", 23, "C++");`**
- Make sure template types support certain features: C++ has concepts now!