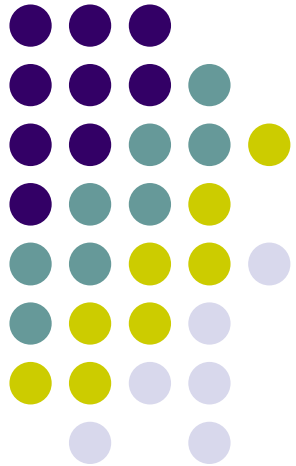
C++

March 2, 2023

Mike Spertus

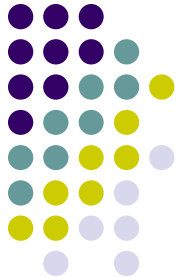
spertus@uchicago.edu



MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

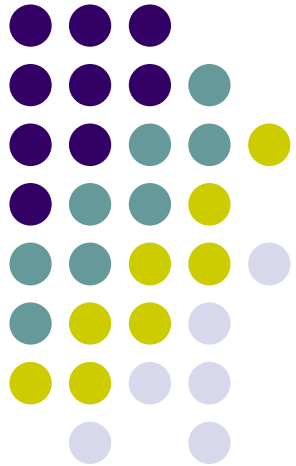
Final



- Next Thursday at normal time
- Open book
- Open notes
- You can look at posted sample files, lecture notes, your past HW submissions, cppreference, general references, and the standard
  - You will definitely want to have ready access to the best practices listed this lecture
- Do not use a compiler
- Do not use anything other than general resources or google for answers to questions
- About half of the final will be similar questions to the homework
- About half will be a code review of a program that I will provide
  - It will violate many of the rules listed later in this lecture, so be sure to check them against the program
- If you are doing a final project instead of the final, it is due on Gradescope by Midnight next Thursday

New and Old Best Practices

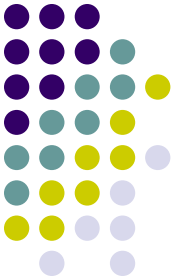
And odds and ends that we've missed



Best practices around default members

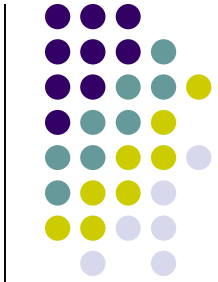
- If you do nothing, the compiler will define the following for you implicitly
  - Default constructor (0-args) default constructs all bases and members
  - Copy constructor copies all bases and members
  - Copy assignment copies assigns all bases and members
  - Destructor destroys all bases and members
  - Move constructor moves all bases and members
  - Move assignment move assigns all bases and members

When should you define your own?



- As we have seen, sometimes the default definitions aren’t what you want
  - Classes designed to be inherited from should have a virtual destructor
  - Sometimes we need a deep copy like btree
- There are some best practices for keeping things consistent

The rule of 3



- If you define any of “copy constructor,” “copy assignment operator” or “destructor,” then define them all
- The point is that if, for example copy construction has custom behavior
  - btree b2(b1); // Deep copies
- Then copy assignment should also
  - b3 = b2; // Should also deep copy

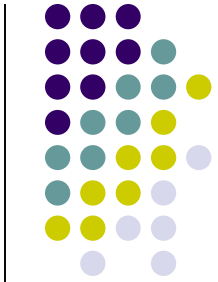
The rule of 5

- If you define a copy constructor, then move constructors are not created by the compiler
  - The compiler thinks that if “copying every member” isn’t the right way to copy, then it’s unsafe to assume that “move every member” is the right way to move
  - Usually it is not harmful not to have move constructors because then you just copy instead
  - But maybe you wanted them
- Rule of 5: If you use the rule of 3, think hard about whether you should define move constructors and move assignment operators or if you’re happy with them just being copies

Guard against multiple inclusion

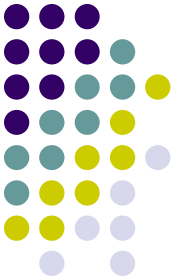
- If the same header is included multiple times, you may get multiply-defined symbol errors
- The following preprocessor idiom prevents that from happening
  - Sometimes the preprocessor is helpful!
- ```
  #ifndef FOO_H
  #  define FOO_H
  ...
  #endif
  ```
  ```

Always put headers in a namespace



- Also use an “#ifndef …” to guard against multiple inclusions
- ```
  #ifndef FOO_H
  #  define FOO_H
  namespace mpcs51044 {
  int f();
  ...
  }
  #endif
  ```
  ```

Never “use” a namespace in a header



- Leaks entire namespace to any file that includes the header.
- E.g., when in a header file, say
  `using std::accumulate;`
  instead of
  `using namespace std;`
  or just explicitly call `std::accumulate` without a `using` statement at all
- When in a “.cpp” file, choose whichever you prefer.

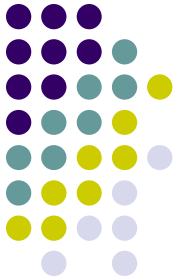
Prefer the C++ versions of standard C headers

- `#include <stdio.h> // Bad`
  `#include <cstdio> // Better`
- The C versions will sort-of work, but the C++ versions will more properly define signatures, so overload resolution, type-checking, etc. will be more robust
- If you have a C header with no C++-specific version (e.g., unistd.h), then of course use the C version

Understand the difference between virtual and nonvirtual

- Virtual methods are based on the dynamic type of the object
- Non-virtual methods use the static type
- This will be on the final
- Let’s review an example from lecture 2

Virtual vs. Non-virtual method

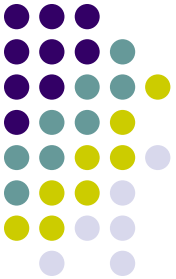


```
struct Animal {
  void f() { cout << "animal f"; }
  virtual void g()  { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f() { cout << "gorilla f"; }
  void g() { cout << "gorilla g"; }
   void h() { cout << "gorilla h"; }
};
void fun() {
   unique_ptr<Gorilla> g = make_unique<Gorilla>;
   Animal &a = *g;
   a.f(); // Not virtual: Animal's f
   a.g(); // Virtual: Gorilla's g
   a.h(); // Error: h is not in animal
   (*g).f(); (*g).g(); (*g).h(); // Gorilla's f, g, and h
}
```

Prefer C++-style casts to C style casts

- A *a = (A *)&b; // bad
- A *a = dynamic_cast<A *>(&b);

Prefer C++-style casts to C style casts -- Rationale



- Let's look at two cases where they differ

```
struct X {...};
struct Y {...};
X *xp = new X;
Y *yp = (X *)(xp); // Nonsense
yp = dynamic_cast<Y *>(xp); // 0 shows cast failed
```

- C++-style casts are better when they disagree

Where should const go?

- const int and int const mean the same thing
- Which is better is hotly debated
- There is even swag for advertising your preference

Put const and volatile after type names

- "`int const`" is better and more consistent than "`const int`"
- The Core Guidelines and C++ Standard disagrees
- Nevertheless, you will be able to get three points on the final for this
- Dan Sachs' ACCU "Truthiness" keynote argues this is the only rational conclusion one can reach, as it is both more logical and studies show that it leads to fewer bugs.
  - What does "`const int *`" mean?
  - Pointer to a constant `int`
  - But it seems just as reasonable to say that it's constant pointer to a modifiable `int`
  - "`int const *`" can't be misinterpreted
  - "`int * const`" can't be misinterpreted

Use const appropriately

- Const methods should be const
- Const & arguments should be const
- The “const” keyword should go after the type
- ```
  class A {
  public:
    void f(int const &i) const;
  };
  ```
  ```

Use const appropriately-rationale

- Ignoring const is no longer an option
- ```
  int seven() { return 7; }
  void pr_int(int &i) { cout << i; }
  void pr_int_const(int const &i) { cout << i; }
  pr_int(7); // Error
  pr_int(seven()); // Error on newer compilers
  pr_int_const(seven()); // OK
  ```
  ```

Use `nullptr` instead of 0 to indicate a null pointer

- C++ adds a new literal `nullptr` of type `nullptr_t` that represents (surprise) a null pointer. Automatically converts to pointer types (and bool)

```
void f(char *) { /* … */ }
void f(int) { /* … */ }

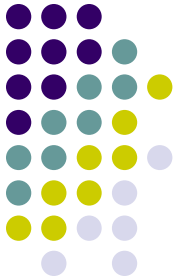
f(0); // OK. Calls f(int)
f(nullptr); // OK. Calls f(char *)
```

- Always prefer the type-correct `nullptr` over the type-incorrect `0` or `NULL` to avoid calling the wrong function/method.

Define symmetric binary operators as global functions

- Don't use the member form of `operator+()`
  - Because both arguments should be treated the same
- However, do define `operator+=()` as a member
  - We don't want to += to try to assign to a compiler-generated temporary

Think about types inferred by templates



- What does this print?

```
double dp[] = { 0.1, 0.2, 0.3 };
cout << accumulate(dp, dp + 3, 0);
```

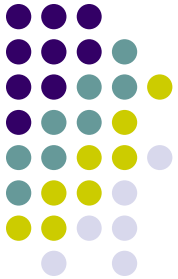
Think about types inferred by templates

- If you're accumulating doubles with `std::accumulate` use an initial value of `0.0` instead of `0`
  - Or you'll accumulate integers
- E.g.,

```
    double dp[] = { 0.1, 0.2, 0.3 };
  cout << accumulate(dp, dp + 3, 0);
```

  (surprisingly) prints `0`
- Probably wanted `cout << accumulate(dp, dp + 3, 0.0)`

Think about types inferred by templates



- The following code to find the maximum length of a collection of strings unexpectedly always prints 0. Why doesn't it work? How can you fix it?

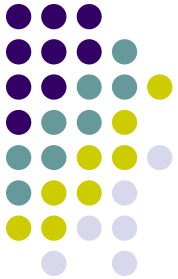
```
struct maxlenftn {
    maxlenftn() { maxlen = 0; }
    void operator()(string s) {
            maxlen = max(maxlen,s.size());
    }
    string::size_type maxlen;
};

int main() {
    vector<string> names{"Spertus", "Lemon", "Golden", "Melhus"};
    maxlenftn maxf;
    for_each(names.begin(),names.end(),maxf);
    cout << maxf.maxlen << endl;
    return 0;
}
```

Explanation

- `maxlenftn` is passed by value, so a copy of it is updated, leaving the original unchanged
- See possible fixes in `for_each.cpp` on Canvas from when we assigned this homework

Beware of Dependent base classes

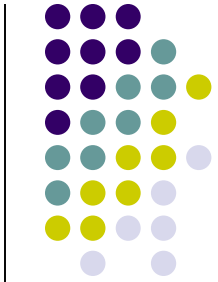


- What does the following print?

```
#include <iostream>
using namespace std;

int f() { return 0; }
template<class T>
struct C : public T {
    C() { cout << f() << endl; }
};
struct A {
    int f() { return 1; }
};
int main()
{
    C<A> c;
}
```

Dependent base classes: Surprising answer



- Microsoft Visual C++ used to print 1
- g++ prints 0
- g++ is correct
- T is a “dependent base class”
  - A base class that depends on the template parameter
- Symbols are not looked up in dependent base classes, so templates are not surprised by unexpected inheritance
  - The "fragile base class" problem

Correct use of dependent base classes

- To see symbols in a dependent base class, reference it explicitly:

```
template<class T>
struct C : public T {
    C() { cout << T::f() << endl; }
};
```

- Alternatively

```
template<class T>
struct C : public T {
    using T::f;
    C() { cout << f() << endl; }
};
```

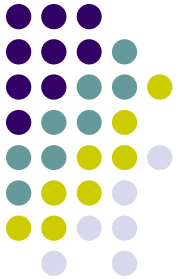
- Another choice

```
template<class T>
struct C : public T {
C() { cout << this->f() << endl; }
};
```

- If you want the global symbol:

```
template<class T>
struct C : public T {
    C() { cout << ::f() << endl; }
};
```

Watch out for method hiding

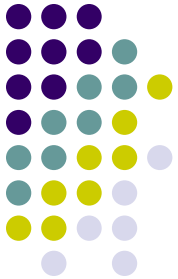


```
struct B {
  void f(bool i) { cout << "bool" << endl; }
};

struct D : public B {
  // Fix with "using B::f"
  void f(int b) { cout << "int" << endl; }
};

int main()
{
  D d;
  d.f(true); // Prints "int"
}
```

Use `override` and `final` to indicate intent

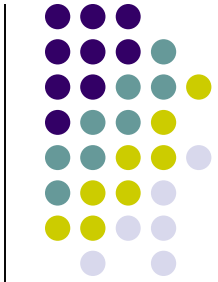


- ```
  struct Base {
    virtual void func() = 0;
    virtual void mispelledFunc() = 0;
  };
  struct Derived : public Base {
    virtual void func() override final {}
    // This will give a useful error
    // because we aren't actually
    // overriding
    virtual void misspelledFunc() override {}
  };
  struct MostDerived : public Derived {
    // Error! Can't override final
    virtual void func() { /*...*/}
  };
  ```
- This will catch a lot of "method hiding" errors
  ```

Throw exceptions by value catch them by const &

```
struct MyException : public exception {
  MyException(string s)
    : myS("My "+s), exception(s) {}
  virtual char const *override what() {
    return myS.c_str();
  }
  string myS;
};
void f() {
  try {
    throw MyException("foo");
  } catch (exception e) { // Bad!
//} catch (exception const &e) { // Better
    cout << e.what(); // May crash due to slicing
}
```

Never have a destructor throw an exception

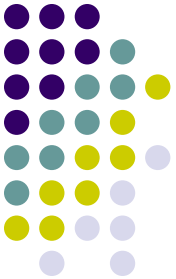


- Does the following catch "`In A`" or "`in f`"?

```
struct A {
  ~A() { throw runtime_error("In A"); }
};
void f()
{
  try {
    A a;
    throw runtime_error("in f");
  } catch (exception const &) {
  }
}
```

- No good answer, so the runtime just calls `std::terminate` to end your program

Use virtual destructors when you inherit



- ```
  class A {
  public:
    // virtual ~A() = default;
  };
  class B : public A {
  public:
    ~B() { ... }
  };
  A *ap = new B;
  delete ap; // Doesn't call B's destuctor
  ```
  ```

Prefer templates to macros

- e.g., `min` should be a template but Microsoft Visual C++ once defined it as a macro
- ```
  #define bad_min(x, y) x<y? x: y;
  int i = 3;
  bad_min(5, i++); // increments i twice!
  2*bad_min(3, 5); // Returns 5!
  ```
- Even figuring out why they fail is tricky
- If we had used a function template, we wouldn't even have needed to worry
  ```

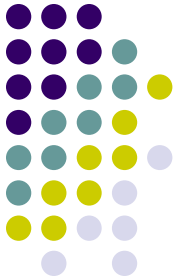
Don’t make tricky assumptions about order of evaluation

```
struct S {
  S(int i) : a(i), b(i++) {
    f(i,i++) // Undefined behavior
  }
  int b;
  int a;
};
```

Remember that primitive types have trivial “constructors”

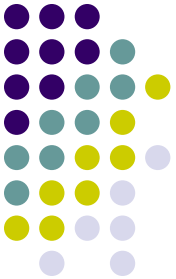
```
void
f()
{
      int i;
    /* int i{}; // Fix with */
      cout << i; // i contains garbage
}
```

Don’t return a reference/pointer to a local variable



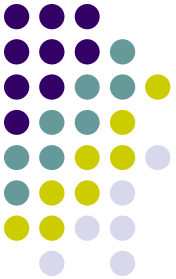
- ```
  int &
  f()
  {
      int i = 3;
      return i; // Bad!
  }
  ```
  ```

Best practice—Prefer range member functions to their single-element counterparts



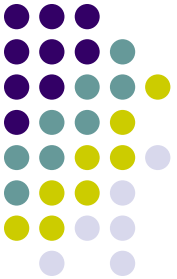
- Item 5 of Meyer's Effective STL
- Given two vectors, v1 and v2, what's the easiest way to make v1's contents be the same as the second half of v2's?
  - Don't worry whether v2 has an odd number of elements

Worst (but common)



- ```
  vector<Widget> v1, v2
  ...
  for(vector<Widget>::const_iterator ci
        = v2.begin() + v2.size()/2;
      ci != v2.end();
      ++ci) {
    v1.push_back(*ci);
  }
  ```
  ```

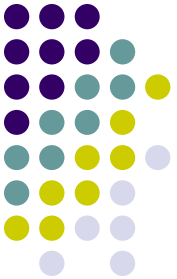
Better



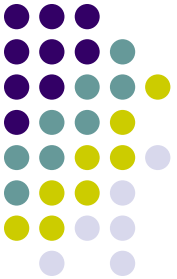
- ```
  copy(v2.begin() + v2.size()/2,
       v2.end(),
       back_inserter(v1));
  ```
  ```

Better yet

- ```
  v1.resize(v2.size() - v2.size()/2);
  copy(v2.begin() + v2.size()/2,
       v2.end(),
       v1.begin());
  ```


  ```

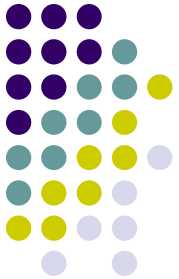
Even better



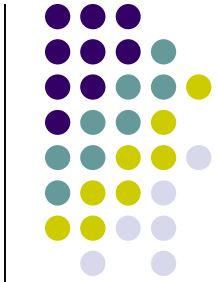
- ```
  v1.insert
      (v1.end(),
       v2.begin() + v2.size()/2,
       v2.end());
  ```
  ```

Best

- ```
  v1.assign(v2.begin() + v2.size()/2,
            v2.end());
  ```


  ```

Best Practice: Prefer empty() to size() == 0



- Suppose `l` is a `list<int>`
- Which is better?
  - `if(l.empty()) { ... }`
  - `if(l.size() == 0) { ... }`
- Prefer the `l.empty()`
- Calculating `size()` can take a long time
- Effective STL Item 4

Always use a smart pointer to manage the lifetime of an object

- unique_ptr if it has only one owner shared_ptr if it has multiple owners
- ```
  Foo *fp = new Foo; // Bad
  // stuff
  delete fp; // May be missed if exception occurred
  ```
- More generally, use RAII to ensure resources get destroyed when they are no longer needed
  ```

Avoid using “new …”

- The problem with saying “`new A()`” is that it returns an owning raw pointer to the new object, violating the preceding best practice
- Prefer using `make_unique` and `make_shared` instead
- `auto ap = make_unique<Foo>(); // Best`

Use RAII to manage locks

- Just like using smart pointers for objects, use a scoped locking class whose destructor releases the lock to make sure locks get released even when exceptions bypass normal control flow
  - Typically, this means to use the std::scoped_lock class, like we do in the false sharing example
  - At more than one job, I (Mike) debugged critical customer defects because manual unlocking code was bypassed by an exception.
    - Moral: Don’t rely on manual unlocking code!

Lock ordering

- If you want to avoid deadlocks, you want to acquire locks in the same order!
  - Suppose thread 1 acquires lock A and then lock B
  - Suppose thread 2 acquires lock B and then lock A
  - There is a window where we could deadlock with thread 1 owning lock A and waiting for lock B while thread 2 owns lock B and is waiting for lock A forever
- The usual best practice is to document an order on your locks and always acquire them consistent with that order
- Also, std::scoped_lock will always get multiple locks without deadlocking as described previously
- See
  http://www.ddj.com/hpc-high-performance-computing/204801163

Memory model best practices

- Here are the takeaways
  - Try to avoid sharing data between threads except when necessary
  - When you share data between threads, always use locks or atomics to ensure both threads have a coherent view of the shared data
- A good reference
  - Boehm, Adve, "*You Don't Know Jack about Shared Variables or Memory Models: Data Races are Evil*" Communications of the ACM 55, 2 Feb. 2012
  - http://queue.acm.org/detail.cfm?id=2088916

Make mutex members mutable

```
struct A {
  int f() const {
    lock_guard<mutex> lck(mtx);
    return i + j;
  }
  int i;
  int j;
  // So const methods can lock
  mutex mutable mtx;
};
```

Prefer using to typedef

- A program might have the following line:
  - `using ConfigVariables =`
    `map < string, variant<string *, bool *, double *>>;`
- We could also have said
  - `typedef`
    `map < string, variant<string *, bool *, double *>>`
    `ConfigVariables;`
- However, typedef doesn't work with templates, has trouble with recursive types, etc., so it is best to just get in the habit of using `using` ☺

Constrain your templates

- If your templates just have typename parameters
- Your user will not be protected from using incorrect types for template arguments and get
  - an incomprehensible error message when something goes wrong later, or
  - even worse, a silently incorrect answer
- Let’s look at an example

Three approaches

- Bad – Silently gives crazy results with doubles
  - template<typename T>
    T gcd(T a, T b) {
      return (b == 0) ? a : gcd(b, a – b*(a/b));
    }
- C++20 Concepts – Good but needs C++20
  - template<integral T>
    T gcd(T a, T b) {
      return (b == 0) ? a : gcd(b, a – b*(a/b));
    }
- Pre-C++20 – Use SFINAE
  - template<typename T, typename = enable_if_t<is_integral_v<T>>>
    T gcd(T a, T b) {
      return (b == 0) ? a : gcd(b, a – b*(a/b));
    }

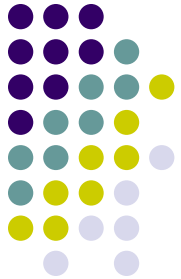
Functions that take in text should use `string_view`

- A string view can understand the underlying region of memory in both a std::string and C string
  - ```
    int f(string_view const s);
    string str("foo");
    auto a = f(str) + f("bar") + f("baz"s); // No copying!
    ```
  - Under the hood it is a pointer and a length, so you can even say `f({"foobar", 3})` to just see `"foo"`
    ```

The most vexing parse

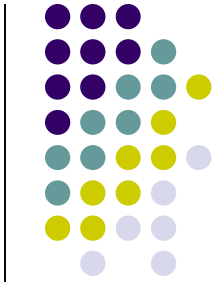
- Sometimes, when you are trying to work with complex types like in the homework, you can confuse the compiler
  - Not just yourself ☺
- This is Item 6 of Scott Meyers' Effective C++
- ```
  ifstream dataFile("ints.dat");
  list<int>
    data(istream_iterator<int>(dataFile),
         istream_iterator<int>());
  ```
- What does this look like it does?
- What does it do?
  ```

Background



- `int f(double d); // Declares function double -> int`
- `int f(double (d)); // Also function double -> int`
- `Foo f(istream_iterator<int>(datafile));`
  `// Oops! Function istream_iterator<int> -> Foo`
- `int f(double(*p)()); // function double(*)()->int`
- `int f(double p()); // Also function double(*)()->int`
- `// Can always omit parameter name`
  `int f(double()); // Again function double(*)()->int`
- `Foo f(istream_iterator<int>());`
  `// Oops. Function istream_iterator(*)() -> Foo`

Solutions (C++98)

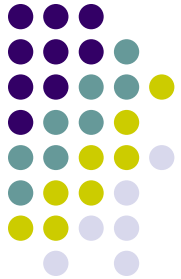


- Could use parentheses to disambiguate
- ```
  ifstream dataFile("ints.dat");
  list<int>
  data((stream_iterator<int>(dataFile)),
       istream_iterator<int>());
  ```
- Could be more explicit
- ```
  istream_iterator<int> dataBegin(dataFile);
  istream_iterator<int> dataEnd;
  list<int> data(dataBegin, dataEnd);
  ```
- These are pretty unnatural, can we do better?
- More examples in https://en.wikipedia.org/wiki/Most_vexing_parse

Uniform initialization

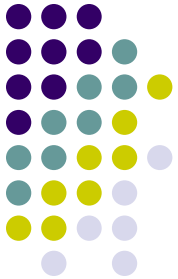
- Bjarne Stroustrup wanted to simplify initialization according the the following principles (http://www.stroustrup.com/list-issues-2.pdf)
  - {...} can be used for every initialization
  - {...} implies direct initialization
  - {...} implies is non-narrowing
- ```
  // Now works fine
  ifstream dataFile("ints.dat");
  list<int>
    data{istream_iterator<int>{dataFile},
         istream_iterator<int>{}};
  ```
- These principles are not fully achievable in practice
  - E.g., why can't I initialize a map as follows?
    `map<int, string> mis = { {4, "abc"s}, {4, "def"s} };`
- Let's look at the paper and some examples for some details
  ```

Namespace versioning



- Namespaces are a great way of versioning APIs
- This way code can say they are using v1 functions and interfaces
- We also want to be able to say "I always want to be on the latest version"

Namespace versioning



- ```
  // MyLib.h
  namespace mpcs51044 {
     namespace v1 {
        int f() { // Original way of calculating f }
     }
     namespace v2 {
        int f() { // Modern efficient way }
     }
     using namespace v2; // Make v2 the default
  }
  ```
- ```
  // MyMissionCriticalProgram.cpp
  #include "MyLib.h"
  auto x = mpcs51044::v1::f();
  ```
- ```
  // MyYoloProgram.cpp
  #include "MyLib.h"
  auto x = mpcs51044::f(); // Latest version
  ```
  ```

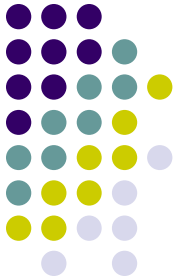
Remember 1st rule of optimization (see case study)

Don't

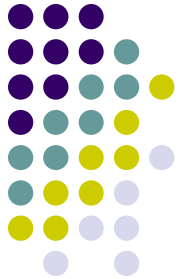
A great place to go for more C++ Best Practices

CORE GUIDELINES

- https://github.com/isocpp/CppCoreGuidelines
- This is the primary community-based repository of C++ best practices



REVIEW OF IMPORTANT CONCEPTS



WEEK 1 (OVERVIEW)

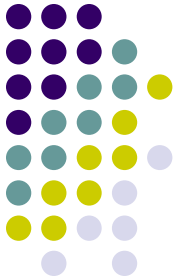
C++ is (partially) a standard

- While there is a carefully worded standard for C++
- Many aspects of C++ are not standardized mostly to better match the hardware they run on
  - Whether `char` is signed, bits in an `int`, etc.
- Standardized > implementation defined > undefined
- Avoid nonstandard vendor-specific extensions to C++ (Microsoft and g++ have quite a few)

C++ is a lightweight abstraction language

- That was the winner when Bjarne Stroustrup moderated a discussion on the C++ elevator pitch
- This is a mouthful, but I think the idea is right
- Programming languages typically fall into a tradeoff between being good for programmers (i.e., abstract) or good for computers (i.e., low-memory. fast, and low-level)
  - Abstract but slow/no low-level programming: Python, Java, JavaScript,…
  - Efficient but not abstract: C, Assembler,
- The problem is, that we need both as performance vs abstraction constantly looms over almost all programming decisions
  - “Do I want to do it the clear modular, maintainable way or the performant way?”
- Unlike almost any other language, C++ gives you the best of both world, allowing you to create powerful abstractions that are lightweight by having all the abstraction “compiled-away” during the build using “templates”
- In other words, there is no performance penalty associated with using/creating abstractions

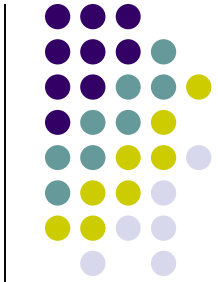
Hello (again) World



```
#include<iostream>
#include<format>
using namespace std;

int
main()
{
    string name;
    cout << "What's your name? ";
    cin >> name;
    cout << format("Hello, {}!\n", name);
    return 0;
}
```

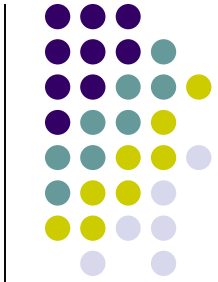
Defining functions



```
int square(int n)
{
  return n*n;
}

auto square(double n)// OK to have two functions with
{                    // same name (overloading) as long
  return n*n;        // as compiler can tell which you
}                    // mean by context
                     // The type "auto" means the compiler
int main() {         // should perform type inferencing
  cout << square(2) + square(3.1416);
  return 0;
}
```

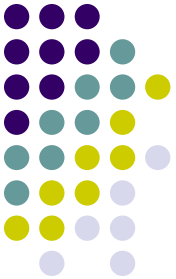
Function templates



```
auto square(auto n)
{
   return n*n;
}

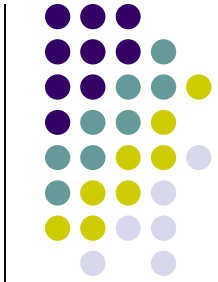
int main()
{
   return square(2) + square(3.1416);
}
```

Function templates (pre-C++20 notation)



```
template<typename T>
auto square(T n)
{
  return n*n;
}

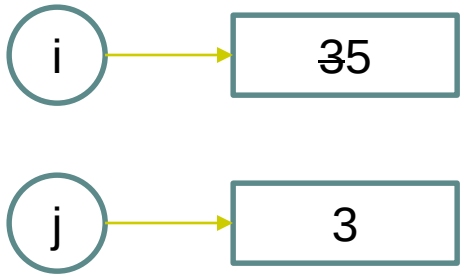
int main()
{
  return square(2) + square(3.1416);
}
```



LECTURES 2 AND 3
OBJECTS, CLASSES, AND THEIR LIFETIMES

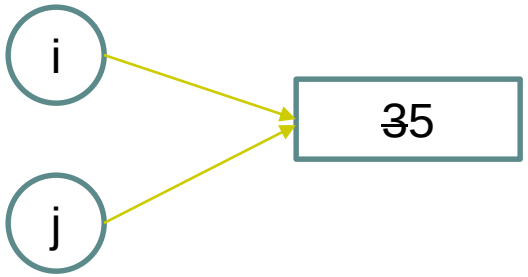
Copying and assignment

- In C++, when we do an assignment, the object is copied
- ```
  int i = 3;
  int j = i;
  i = 5;
  cout << j; // Prints 3
  ```
- https://godbolt.org/z/_GYLWx
- In this example, both `i` and `j` have their own storage associated with them in the running program
- **Warning:** In Java and Python, built-in types like int are copied, but user-defined types are not copied by assignment but are instead shared
  - cf next slide


  ```

References

- Suppose we wanted j to refer to the same data in memory as i, instead of a copy
- We can do that by creating j as a *reference* to an existing object (`int &`)
- ```
  int i = 3;
  int &j = i;
  i = 5;
  cout << j; // Prints 5
  ```
- https://godbolt.org/z/z2GDSr


  ```

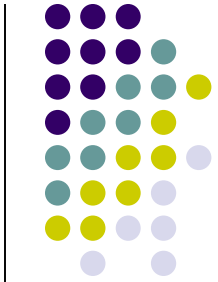
A unique_ptr

- Gives you access to the data in the object
  - When you need a reference to the object managed by the unique_ptr, use the * operator
  - ```
    unique_ptr<int> ui = make_unique<int>(5);
    cout << *ui; // Prints 5
    ```
- Manages the lifetime of the object
  - When the unique_ptr goes away, it will automatically free up the memory of the object that it is managing
    ```

Updating a unique_ptr

- If you bind a `unique_ptr` to point to a new object, it will free up the old one before it starts to manage the new one
- ```
  unique_ptr<int> ui = make_unique<int>(5);
  cout << *ui; // Prints 5
  ui = make_unique<int>(2);
  cout << *ui; // prints 2
  ```
- The `unique_ptr` automatically releases the memory of the first object (the one with value `5`) before it starts managing the new object (the one with value `2`)
  ```

Transferring ownership



- In our example program later today, we will need to transfer ownership from one `unique_ptr` to another
- This is a little tricky
- Assignment doesn't work
  - `up2 = up1; // Oops!  Two "unique" owners`
- Correct solution. Move from `up1`
  - `up2 = move(up1); // up2 is owner. Don't use up1`
  - For Rustaceans, this is like using assignment to transfer ownership in Rust
- We will learn more about moving soon
- But for now, you can treat it as a magic word

A class is how you create your own types

- Classes have member functions (methods) and data members (fields)

```
struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  // Method to calculate the student's grade
  double grade() const {
    return (midterm + final + median(homework))/3;
  }
};
```

Accessing methods and fields

- Use the . operator
- ```
  Student_info s;
  s.name = ”Mike”;
  s.midterm = 70;
  s.final = 85;
  s.homework.push_back(60);
  s.homework.push_back(75);
  cout << s.grade();
  ```
  ```

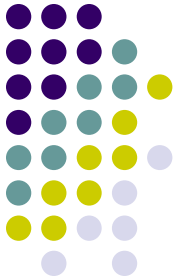
Static vs Dynamic types

- As we mentioned above, a program uses expressions to refer to objects in memory
- The static type is the type of the expression
  - Known at compile time
- The dynamic type is the type of the actual object referred to by the expression
  - May be knowable only at run-time
- Static and dynamic type generally only differ due to inheritance

(Single) inheritance

- One can use inheritance to model an “isA” relationship
- `struct Animal { /* ... */ };`
- `class Gorilla : public Animal {…};`
- This means that a `Gorilla` “isA” `Animal` and can be referred to by `Animal` references

Static type vs Dynamic type

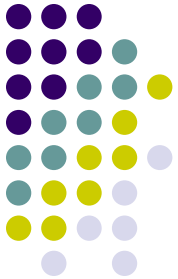


```
int i = 5; // S = int, D = int
Gorilla g; // S = Gorilla, D = Gorilla
Animal &a = g; // S = An&, D = Gor
Animal a2 = g; // Oops! Can't copy a Gorilla
                // into an Animal
unique_ptr<Animal> ua = make_unique<Gorilla>();
// Static type of *ua is Animal but Dynamic is Gorilla
ua = make_unique<Falcon>();
// Now S = Animal, D = Falcon
```

Virtual vs. non-virtual method

- A virtual method uses the dynamic type
- A non-virtual method uses the static type

Virtual vs. Non-virtual method



```
struct Animal {
  void f() { cout << "animal f"; }
  virtual void g()  { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f() { cout << "gorilla f"; }
  void g() { cout << "gorilla g"; }
  void h() { cout << "gorilla h"; }
};
void fun() {
   unique_ptr<Gorilla> g = make_unique<Gorilla>;
   Animal &a = *g;
   a.f(); // Not virtual: Animal's f
   a.g(); // Virtual: Gorilla's g
   a.h(); // Error: h is not in animal
   (*g).f(); (*g).g(); (*g).h(); // Gorilla's f, g, and h
}
```

Passing arguments

- In C++, there are three ways to pass arguments to a function
- **By value**
  - `void f(X x);`
  - `x` is a copy of the caller's object
  - Any changes `f` makes to `x` are not seen by the caller
- **By reference**
  - `void g(X &x);`
  - `x` refers to the caller's existing object without creating a new copy
  - Any changes `g` makes to `x` also change what the caller sees
- **(No longer a) Spoiler: By move**
  - `void h(X &&x);`
  - We will learn about this soon

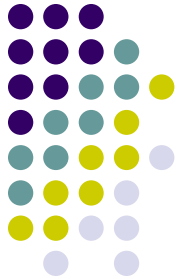
invoking a constructor to create an object

```
// construction.cpp on chalk
#include<initializer_list>
using std::initializer_list
struct A {
  A(int i, int j = 0); // #1
  explicit A(int i);  // #2
};
struct B {
  B(int i, int j); // #3
  B(initializer_list<int> l); // #4
};
int main() {
  A a0 = 3; // A(3, 0) #1
  A a1(3);  // A(3) #2
  A a1{3};  // A(3) #2. Uniform init: best practice but
  B b0(3, 5); // #3
  B b00 = {1, 2, 3, 4}; // #4
  B b1 = { 3, 5}; // #4
  B b2{3, 5}; // #4 Initializer list is preferred
}
```

Default constructor

- If you don't declare any constructors, C++ will create a no-argument constructor for you
- Even though S was declared as
  - struct S { int i; double d; };
- The compiler generates as if you wrote
  - struct S {
    S() {}
    int i;
    double d;
    };

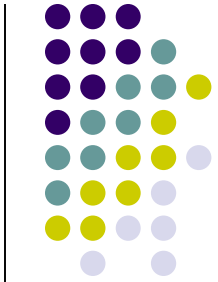
Copy constructors



- The compiler implicitly generates a copy constructor that copies all of the base classes and members as if you wrote
- struct S {

```
  S(S const &s) : i(s.i), d(s.d) {}
  int i{}; double d = 1.2;
};
```

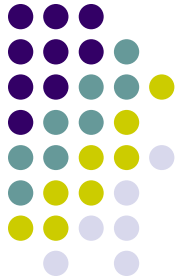
Move constructors



- If you don't define your own default, copy, or move constructor (if all of those have default behavior, the compiler figures move does to)
- The compiler generates a move constructor that moves all the members and base classes
- ```
  struct S {
    S(S &&s) : i(move(s.i)), d(move(s.d)) {}
    int i{}; double d = 1.2;
  };
  ```
  ```

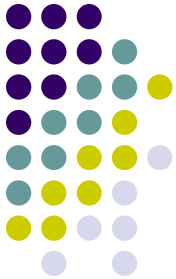
Copy and move assignment

- The compiler generates similar copy and move assignment (operator=()) operators as well as constructors
- Note that you can explicitly define or delate any of these constructors, destructors, or assignment operators if you don't want what the language provides by default



TYPE CONVERSIONS

Implicit conversions

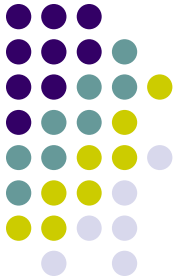


- Built-in
  - ```
    int i = 78Ø;
    long l = i;
    char c = 7;
    char c = i; // No warning, but dangerous!
    ```
- Polymorphism
  - `unique_ptr<Animal> ap = make_unique<Dog>("Champ");`
  - `void f(Dog &dr) { Animal &ar = dr; }`
- User-defined
  - Constructors
  - Operator overloading
- "Standard Conversions"
  - Defined in clause 4 of the standard
    ```

Constructors and typecasts

```
struct A {
  A();
   A(int i);                  // Creates a type conversion
   A(int i, string s);
   explicit A(double d); // Does not create a conversion
};
A a;
A aØ(1, ”foo”);
A aa = { 1, ”foo”};
A a1(7); // Calls A(int)
a1 = 77;// ok
A a3(5.4); // Calls A(double)
a3 = 5.5; // Calls A(int)!!
```

Type conversion operators



```
struct seven {
   operator int() { return 7; }
};
struct A { A(int); }
int i = seven();
A a = 7;
A a = seven(); // Ill-formed, two user-defined
// conversions will not be implicitly composed
```

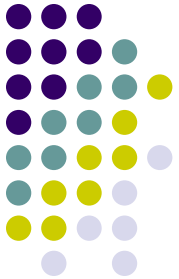
Explicit conversions

- Old-style C casts (Legal but bad!)
  - `char *cp f(void *vp) { return (char *)vp; }`
- New template casting operators
  - `static_cast<T>`
    - Like C casts, but only makes conversions that are always valid. E.g, convert one integral type to another (truncation may still occur).
  - dynamic_cast<T&>
    - Casts between reference types. Can even cast a Base& to a Derived& but only does the cast if the target object really is a Derived&.
    - Only works when the base class has a vtable (because the compiler adds a secret virtual function that keeps track of the real run-time type of the object).
    - If the object is not really a T &, dynamic_cast<T&> throws std::bad_cast;
  - reinterpret_cast<T*>
    - Does a bitwise reinterpretation between any two pointer types, even for unrelated types. Never changes the raw address stored in the pointer. Also can convert between integral and pointer types.
  - const_cast<T>
    - Can change constness or volatileness only. There are usually better alternatives

THE C++ OBJECT LIFECYCLE

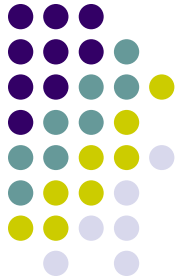
Object duration

- Automatic storage duration
  - Local variables
  - Lifetime is the same as the lifetime of the function/method
- Static storage duration
  - Global and static variables
  - Lifetime is the lifetime of the program
- Dynamic storage duration
  - Lifetime is explicit
  - Created with commands like “make_unique”
  - Programmer controls when it ends
- In all cases, the constructor is called when the object is created and the destructor is called when the object is destroyed



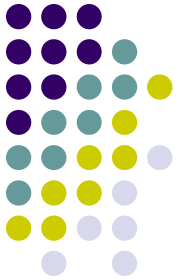
LECTURE 4

Lambdas



- As we saw earlier in this course, besides named functions, you can also call lambdas
- Here is one that squares its integer argument
- `[](int x)->int { return x*x; }`

Lambda return values



- If your lambda does not contain return statements that return different types, the compiler can usually figure out the return type without your saying anything

```
[](int x, int y) {
  int z = x + y; return z + x;
}
```

- If needed, you can explicitly specify the return type with the following notation

```
[](int x, int y) -> int {
  int z = x + y; return z + x;
}
```

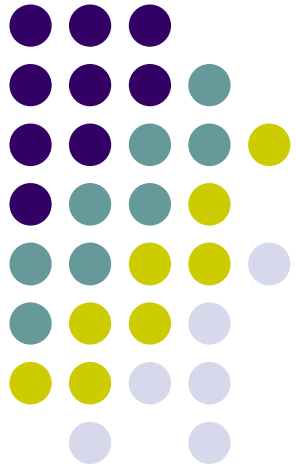
Capture lists

- To capture local variables by reference, use [&]
- To capture local variables by value, use [=]
  - ```
    auto f() {
      int i{3};
      auto lambda = [=](){ cout << i; }; // lambda has a copy of i
      return lambda;
    }
    // Even if the original i has gone away, you can still run lambda
    f()(); // prints 3 even though original i no longer exists
    ```
- C++ annoyingly makes `const` copies, so they may be hard to modify. You can turn this off with the keyword mutable as follows
  - ```
    int i{3};
    auto lambda = [=]() mutable { cout << i++; }; // lambda has a copy of i
    // Note: Only modifies lambda's i, not the original variable
    ```
- You can write fancier capture rules if necessary
  - See https://en.cppreference.com/w/cpp/language/lambda for details

Functors

- If you write your own class with an overloaded operator(), it's called a functor
- This allows you to create stateful objects that can be called like functions
- ```
  struct Accumulator {
    int operator()(int j) { i = i + j; return i }
    int i = 0;
  };
  Accumulator acc;
  cout << acc(1); // Prints 1
  cout << acc(2); // Prints 3 = 1+2
  cout << acc(4); // Prints 7 = 1+2+4
  ```
- As shown in the previous slides, lambdas are often implemented as functors
  ```

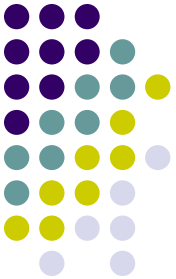
Move Semantics



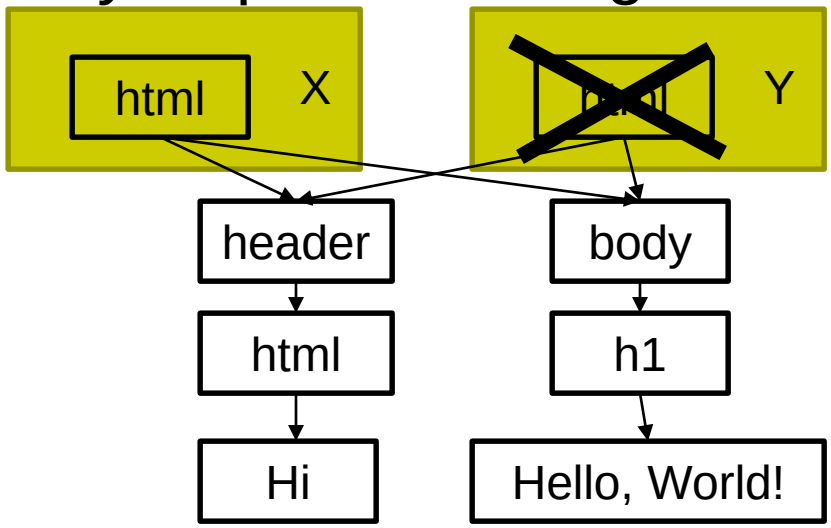
What does it mean to move an object

- Moving y to x
  - x = move(y);
- Means that the value of x is the same as the value y started at
- So one way to do this is by using ordinary copying like
  - x = y;
- However, the difference between a copy and a move is that y is allowed to be changed by the move
  - Subsequent code should not use the value of y

Why move rather than copy? Efficiency



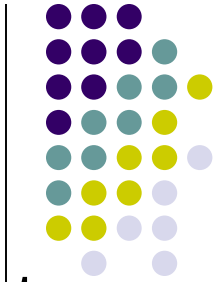
- Since a move can “raid the old object for parts,” it can be a lot more efficient
  - Copying a tree requires copying all of the branches
  - Moving a tree only requires moving the root



How does C++ know whether to move or copy?

- If it is sure that the old object doesn't need its value any more
- One way is if the old object is a temporary value and has no name, so it can't be referred to again
  - In the following example, it is safe to move the unique_ptr returned by make_unique into x, since that is the only place the return value is visible
  - auto x = make_unique<X>(); // x owns the X obj
- The other way is if the programmer uses std::move to say that it is ok to move from
  - auto y = move(x); // Now y owns the X obj

How does move work? Rvalue references



- A reference with && instead of just & can bind to a temporary and move it elsewhere.
- Objects are often much cheaper to “move” than copy
  - For example, deep copying a tree
  - Some objects, like unique_ptr can be moved but not copied
- template<class T>
  void swap(T& a, T& b)// "perfect swap"(almost)
  {
  T tmp = move(a); // could invalidate a
  a = move(b); // could invalidate b
  b = move(tmp); // could invalidate tmp
  }

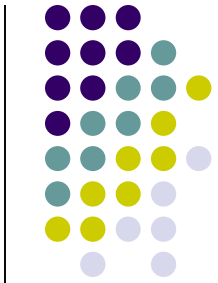
How do I make a type movable?

- First, you don't *need* to make a class movable. It is only a performance optimization because C++ will just copy if there are no move operations
- If moving is more efficient, you should create move constructors and move assignment operators, just like standard library containers do:
- template<class T> class vector {
  // ...
  vector(vector<T> const &); // copy constructor
  vector(vector<T> &&); // move constructor
  vector& operator=(const vector<T>&); // copy assignment
  vector& operator=(vector<T>&&); // move assignment };
- Sometimes, the compiler will automatically generate move constructors and move assignment operators that just move all the members
  - Basically if you don't define a copy constructor/assignment operator or a destructor
- If you want to force the compiler to generate the default move constructor even though it wouldn't normally, you can force that with `default`
- struct S {
  S(S const &); // OK, but stops move constructor generation
  S(S &&) = default; // Gets it back
  /* ... */
  };

Rule of five?

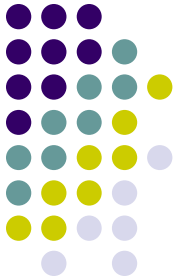
- There is a lot of discussion about whether the rule of 3 should be extended to a “rule of 5,”
  - If you define any of
    - The destructor
    - The copy constructor
    - Copy assignment operator
    - Move constructor
    - Move assignment operator
  - You should review that they all do the right thing because they are all related

- C++11 deprecated some features to better mesh with the rule of 5
  - A proposal (with history) to remove the deprecated features was rejected for C++14. Even though it was rejected it makes interesting and illuminating reading for aspiring language lawyers
    - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3578.pdf
    - Warning: This paper is hard-core. Only recommended if you have substantial C++ experience



TEMPLATES

OO and Templates can solve similar problems



- Consider the following OO code

```
struct Animal {
   virtual string name() = 0;
   virtual string eats() = 0;

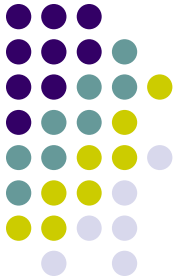
};

class Cat : public Animal {
   string name() override { return "cat"; }
   string eats() override { return "delicious mice"; }

};
// More animals...

int main() {
   unique_ptr<Animal> a = make_unique<Cat>();
   cout << "A " << a->name() << " eats " << a->eats();
}
```

Do we need to use OO?



- Not really

```
struct Cat {
    string eats() { return "delicious mice"; }
    string name() { return "cat";}
};
// More animals...

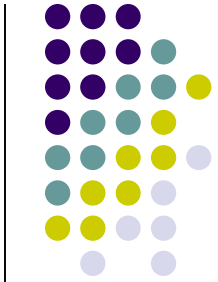
int main() {
    auto a = Cat();
    cout << "A " << a.name() << " eats " << a.eats();
}
```

Concepts

- Concepts play the analogous role for generic programming that base classes do in object oriented programming
- A concept explains what operations a type supports
- The following concept encapsulates the same info as the base class

```
template<typename T>
concept Animal = requires(T a) {
   { a.eats() } -> convertible_to<string>;
   { a.name() } -> convertible_to<string>;
};
```

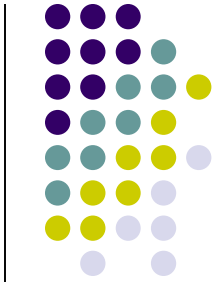
Now, we can ensure that a represents an animal



- With the above concept defined, we can specify that a must satisfy the Animal concept, and the compiler will not let us initialize it with a non-Animal type like House or int

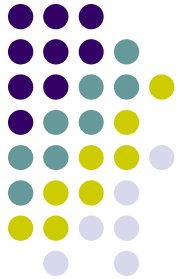
```
int main() {
    Animal auto a = Cat();
    cout << "A " << a.name() << " eats " << a.eats();
}
```

Let's compare



- https://godbolt.org/z/cWc6aM
- As you can see, the definition of `Cat` and the client code in `main()` look very similar in both
- This follows a principle enunciated by Bjarne Stroustrup
  - "Generic Programming should just be Normal Programming"

Specializing and overloading templates

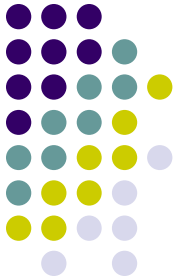


- The “secret sauce” for C++ templates is that if the general “generic” definition of the template isn’t really what you want for a particular set of template parameters, you can override it for that particular case with a specialization
- Think of this as the compile-time analog to object orientation where you also override a more general method in a more specialized derived class

Matrix determinants

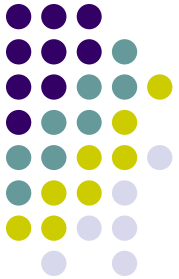
- The determinant is a number that represents "how much a matrix transformation expands its input"
  - Don't worry if you don't understand this
- We will just use the formula to calculate them
- The general formula is here
  - http://en.wikipedia.org/wiki/Laplace_expansion
- But I will give you the special case you need for the homework

Overloading



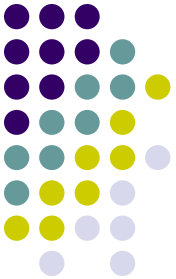
- Function templates can be overloaded
- For example, see OverloadMatrix.h

Full specialization



- A function, class, or member can be fully specialized
- See the definition of Matrix<1,1>::determinant() in Matrix.h

Partial specialization



- Only classes may be partially specialized
- Template class:

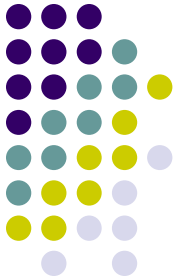
```
template<class T, class U>
class Foo { ... };
```

- Partial specialization:

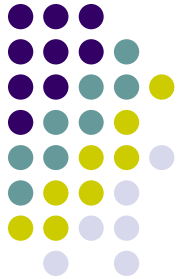
```
template<class T>
class Foo<T, int> {...};
```

- You can tell the second is a specialization because of the <> after the class name

Partial specialization

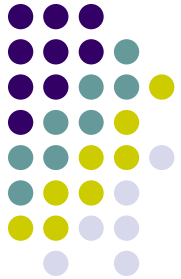


- The partially specialized class has no particular relation to the general template class
  - In particular, you need to either redefine (bad) or inherit (good) common functionality
  - For example, see PSMatrix.h



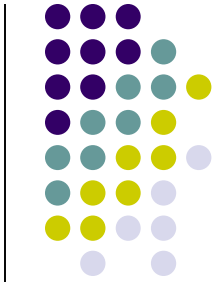
LECTURE 5

static_assert



- What happens if we call determinant() on a non-square matrix?
- Let’s give it a try
- Wow! That was a weird error message
- Can we do better?

static_assert



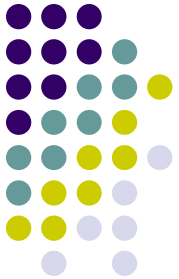
- C++ lets you create a compile-time assertion that prints a nice error message of your choice
- ```
  static_assert(rows == cols,
  "Sorry, only square matrices have determinants");
  ```
- Much better
  ```

*this

- Sometimes a method needs a reference to the object it is a part of
  - `*this`
- In OverloadMatrix, `Matrix::determinant()` uses this to pass the containing `Matrix` to the external `determinantImpl` function

```
template<class T, int h, int w>
T
Matrix<T, h, w>::determinant() const
{
     return determinantImpl(*this);
}
```

- This also comes in handy in the homework for overloading `Matrix::operator+=`



CONSTEXPR

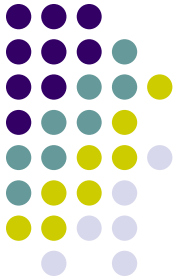
constexpr allows you to do things at compile-time

- Normally we think of programming as a way to write code that runs at runtime
- It is surprisingly common that you need “something” to take place at compile-time instead of runtime
- What do I mean by “something”?
- We will look at several examples over the next few slides

`constexpr` functions

- A constexpr function cannot contain just any code.
  - For example, if it contained a `thread_local` variable, what would that mean at compile-time?
- What is not allowed:
  - Uninitialized declarations
  - `static` or `thread_local` declarations
  - Modification of objects that were not created in the function
    - Or potential modifications by, say, calling a non-constexpr function.
  - virtual methods
  - Non-literal return types or parameters
    - A type is literal if its constructor is trivial or constexpr

Fixing the square example



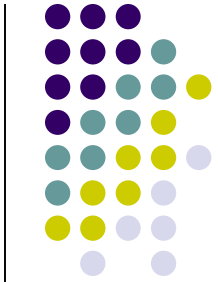
- Now our code runs
- ```
  auto square(int x) constexpr
  { return x*x; }

  Matrix<square(3), square(3)> m;
  ```
  ```

Tuples

- A tuple is an anonymous struct with fields of given types
- You can think of tuples having the same relation to structs as lambdas do to functions
- You can use get<> to access the fields
- ```
  tuple<string, int, double> si("str", 2, 3.5);
  tuple di(2.5, 3, 'c');
  // CTAD: di will be a tuple<double,int, char>
  cout << get<0>(di) // prints 2.5
  cout << get<char>(di); // prints 'c'
  int three = get<1>(di);
  ```
  ```

Returning multiple values

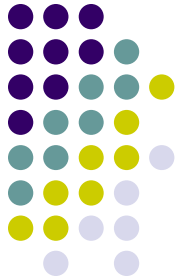


- One natural use for pair and tuple is to let functions return multiple values
  - ```
    pair<int, int> f() {
      return {1, 2}; // ok
    }
    ```
  - ```
    tuple<int, int, char> g() {
      return {1, 2, 'u'}; // OK starting in C++17
    }
    ```
- For more information on why it took until C++17 for the tuple example, see Improving Pair and Tuple (revision 1) by Daniel Krugler
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3739.html
  - Warning: Very advanced

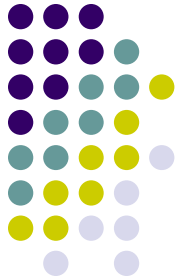
Structured bindings

- Suppose we have a function that returns either a web page or a network error code
- ```
  pair<unique_ptr<Page>, int> getWebPage() {
    /* … */
    if(succeeded)
      return { page, Ø };
    else
      return { unique_ptr<Page>, errCode };
  }
  /* ... */
  auto [ page, err] = getWebPage();
  if(!err) processPage(page)
  else if (err == 4Ø4)  { cout << ”Not found” << endl; }
  else { cout << format(”Unknown error {}\n“, err); }
  ```
  ```

The perfect-forwarding problem

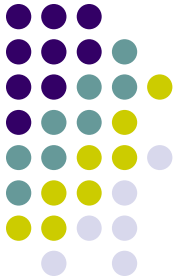


- How can we pass our arguments as-is to another function using that function's signature
- The standard supplies a std::forward function that does exactly that
- ```
  // Perfectly forwards everything
  template<typename T, typename... Args>
  unique_ptr<T> makeUnique(Args &&... args)
  {
    return unique_ptr<T>(new T(forward<Args>(args)...));
  }
  ```
  ```



THREADS

Hello, threads

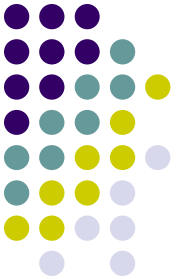


```
#include <iostream>
#include <thread>

void hello_threads() {
    std::cout<<"Hello Concurrent World\n";
}

int
main(){
    // Print in a different thread
    std::thread t(hello_threads);
    t.join(); // Wait for that thread to complete
}
```

Locks

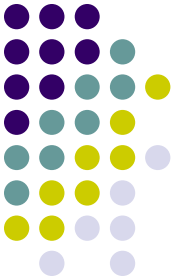


```
std::list<int> some_list; // A data structure accessed by multiple threads
std::mutex some_mutex;  // This lock prevents concurrent access to the shared data structure

void
add_to_list(int new_value)
{
  // Since I am going to access the shared data struct, acquire the lock
  std::lock_guard guard(some_mutex); // CTAD deduces lock_guard<mutex>
  some_list.push_back(new_value);
  // Now it is safe to use some_list. lock_guard destructor releases lock at end of function
  /* ... */
}

bool
list_contains(int value_to_find)
{
    std::lock_guard guard(some_mutex); // Must lock to access some_list
    return std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
}
```

Reader-Writer Locks shared_mutex



```
std::list<int> some_list;
std::shared_mutex some_mutex;

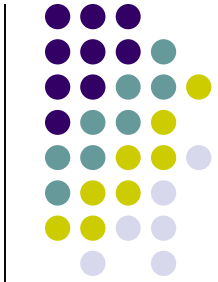
void
add_to_list(int new_value)
{
    std::unique_lock guard(some_mutex); // Unique writer access
    some_list.push_back(new_value);
}

bool
list_contains(int value_to_find)
{
    std::shared_lock guard(some_mutex); // Shared reader access
    return
        std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
}
```

Lock ordering

- If you want to avoid deadlocks, you want to acquire locks in the same order!
  - Suppose thread 1 acquires lock A and then lock B
  - Suppose thread 2 acquires lock B and then lock A
  - There is a window where we could deadlock with thread 1 owning lock A and waiting for lock B while thread 2 owns lock B and is waiting for lock A forever
- The usual best practice is to document an order on your locks and always acquire them consistent with that order
- See http://www.ddj.com/hpc-high-performance-computing/204801163
- Tools like ThreadSanitizer can also help find potential deadlocks and data races
  - https://clang.llvm.org/docs/ThreadSanitizer.html

Sometimes it is hard to fix a lock order



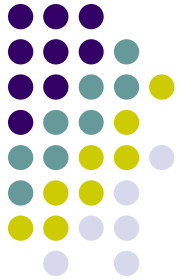
- From http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-7-locking-multiple-mutexes.html
- Consider

```
class account {
 mutex m;
 currency_value balance;
public:
 friend void transfer(account& from,account& to,
                      currency_value amount) {
   lock_guard lock_from(from.m);
   lock_guard lock_to(to.m);
   from.balance -= amount;
   to.balance += amount;
 }
};
```

- If one thread transfers from account A to account B at the same time as another thread is transferring from account B to account A: Deadlock!

C++ provides a solution

- `std::scoped_lock` allows you to acquire multiple locks “at the same time” and guarantees there will be no deadlock,
  - Like magic! (Actually, it will try releasing locks and then acquiring in different orders until no deadlock occurs)
- ```
  class account {
   mutex m;
   currency_value balance;
  public:
   friend void transfer(account& from,account& to,
                        currency_value amount) {
     scoped_lock lck(from.m, to.m);
     from.balance -= amount;
     to.balance += amount;
   }
  };
  ```
  ```



EXCEPTIONS AND RAII

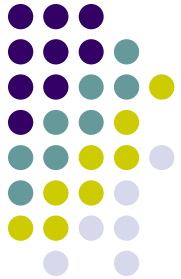
Exceptions

- When something goes wrong, throw an exception
  - This ensures that errors are never inadvertently ignored
- Can throw an exception (any type) with `throw`
  - But usually they should inherit from `std::exception`
- You can catch an exception within a try block with `catch`.
- If you don't catch the exception, it is passed to the caller and then the caller's caller, etc. until it is caught
- Simple example at https://godbolt.org/z/gJ97bS

RAII

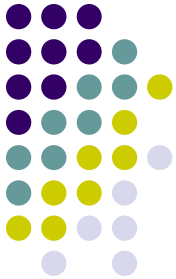
- “Resource Acquisition is Initialization”
- One of the most important idioms in C++
- RAII uses automatic duration objects to manage the lifetimes of dynamic duration resources
- RAII just means that we ensure that an automatic object’s destructor performs any needed cleanup

RAII Classes



- Just of the classes we know about, unique_ptr, shared_ptr, lock_guard, unique_lock, shared_lock, jthread all use their destructor to release the object they manage
- Let's take a look at what can go wrong if we don't use RAII classes

What if we didn't use RAII?



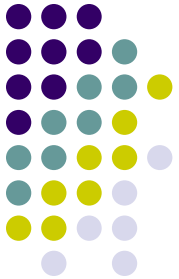
```
std::list<int> some_list;
std::shared_mutex some_mutex;

void
add_to_list(int new_value)
{
   some_mutex.lock();
   some_list.push_back(new_value);
       some_mutex.unlock();
}

bool
list_contains(int value_to_find)
{
   some_mutex.shared_lock();
   auto result =
      std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
       some_mutex.shared_unlock();
       return result;
}
```

Cache-conscious programming (Adapted from Herlihy&Shavit p. 477

- Objects or fields that are accessed independently should be aligned and padded so they end up on different cache lines.
- Keep read-only data separate from data that is modified frequently.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. While a shared counter would cause invalidation traffic, the split counter allows each thread to update its own replica without causing cache coherence traffic.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that is frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.
- If a class or struct contains a large chunk of data whose size is divisible by a high power of two, consider separating it out of the class and holding it with an `unique_ptr` to avoid the Ghostscript problem from the previous slide
- Use a profiling tool like VTune to identify where your cache bottlenecks are



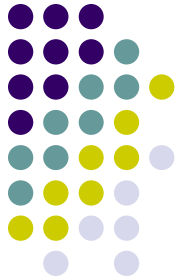
LECTURE 6

Pointers

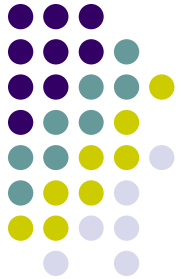
- Pointers to a type contain the address of an object of the given type.
  A *ap = new A();
- What does this mean?
- ap contains the address of an object of type A in memory
- new A() constructs an object of type A in memory and returns its address
- new A() has the same relation to A * as make_unique<A>() has to unique_ptr<A>
- A a;
  ap = &a; // & is the addressof operator

Pointers

- Dereference with *
  A a = *ap;
- -> is an abbreviation for (*_).
  ap->foo(); // Same as (*ap).foo()
- All of these should be familiar from unique_ptr (which intentionally provides a pointer-like interface)
- **Warning:** Unlike unique_ptr, raw pointers do not have destructors and do not manage the lifetime of the object they point to. They simply store its address



LECTURE 7

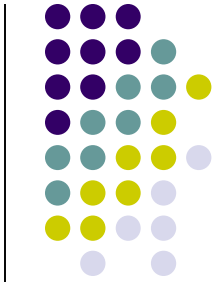


INTER-THREAD COMMUNICATION

Sometimes locks aren't what you want

- Suppose we are trying to implement a "producer/consumer" design pattern
  - Think of this as a supply chain
  - Some threads produce work items that are consumed by other threads
  - Incredibly common in multi-threaded programs
- Typically the producers put work onto a queue, and the consumers take them off
- Locks can allow thread-safe access to the queue
- But what happens if there is no work at the moment?
  - The consumer thread needs to go to sleep and wake up when there is work to do
  - Rather than a lock, you'd like to wait for an "event" stating that the queue has become non-empty

Producer-consumer implementation

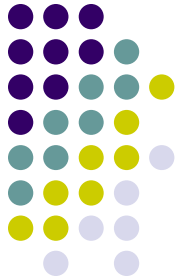


- We will use a couple of new library features
  - `unique_lock`,
    - a richer version of `lock_guard`
  - `condition_variable`
    - “wakes up” waiting threads

Condition variables

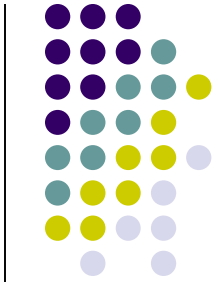
- The C++ version of an event
  `condition_variable cv;`
- You can wait for a condition variable to signal
```
mutex m;
boolean test();
unique_lock<mutex> lck(m);
cv.wait(lck, test);
```
- If the test succeeds, the wait returns immediately, otherwise it unlocks m (that's why we used a `unique_lock` instead of `lock_guard`)
- Once the `condition_variable` signals the waiting thread (we'll see how in a moment)
  - The lock is reacquired
  - The test is rerun (if it fails, we wait again)
    - Protects against spurious wakeups
  - Once the test succeeds, the program continues

Signaling an event is simple



- `cv.notify_one();`
  - Wakes one waiter
    - No guarantees which one
- `cv.notify_all();`
  - Wakes all waiters

Producer-consumer from Williams' *C++ Concurrency in Action*

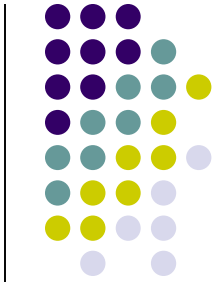


```
mutex m;
queue<data_chunk> data_queue;
condition_variable cond;

void data_preparation_thread()
{
  while(more_data_to_prepare()) {
    data_chunk const data=prepare_data();
    lock_guard lk(m);
    data_queue.push(data);
    cond.notify_one();
  }
}

void data_processing_thread()
{
  while(true) {
    unique_lock<mutex> lk(m);
    cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data=move(data_queue.front());
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Async functions: Running functions in another thread



- It's nice that we can pass arguments to a thread (like we do to functions), but how can we get the thread to return a value back?
- Basically, we want to be able to use threads as "asynchronous functions"

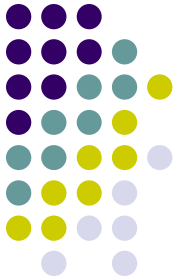
```
std::future<int> f = std::async(func_returning_int);
```

- C++11 defines a `std::future` class template that lets a thread return a value sometime in the future when it's calculation is complete
- One way to create a future is with `std::async`
  - As soon as you create it, it starts running the function you passed it (usually) in a new thread
  - Call `get()` when you want to get the value produced by the function
  - `get()` will wait for the thread function to finish, then return the value

Introducing promises and futures

- Promises and futures let you pass values between threads
- Kind of like a producer/consumer but you are just passing a single item, so you don't need a queue
- A promise lets you pass a value from one thread to another
- Create a paired future and promise
  - `promise<int> p;`
    `future<int> fi = p.get_future();`
- The producing thread stores the value in the promise
  - `p.set_value(7);`
- The consuming thread receives the value when it becomes available
  - `cout << "received " << fi.get() << endl;`

std::future example

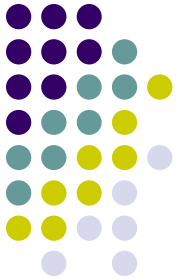


- From Multithreading in C++0x Part 8

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
  // Run calculation in a different thread
  std::future<int> the_answer
            = std::async(calculate_the_answer_to_LtUaE);
  do_stuff();
  std::cout <<"The answer to life, the universe and everything is "
            << the_answer.get()
            << std::endl;
 }
```

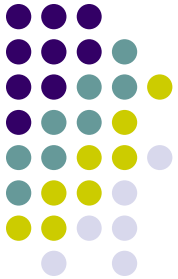


LECTURE 8
"OO" TECHNIQUES

Tuples

- A tuple is an anonymous struct with fields of given types
- You can think of tuples having the same relation to structs as lambdas do to functions
- You can use get<> to access the fields
- ```
  tuple<string, int, double> si("str", 2, 3.5);
  tuple di(2.5, 3, 'c');
  // CTAD: di will be a tuple<double,int, char>
  cout << get<0>(di) // prints 2.5
  cout << get<char>(di); // prints 'c'
  int three = get<1>(di);
  ```
  ```

Structured binding



- Suppose we have a function that returns either a web page or a network error code
- ```
  pair<unique_ptr<Page>, int> getWebPage() {
    /* … */
    if(succeeded)
      return { page, 0 };
    else
      return { unique_ptr<Page>, errCode };
  }
  /* ... */
  auto [ page, err] = getWebPage();
  if(!err) processPage(page)
  else if (err == 404)  { cout << "Not found" << endl; }
  else { cout << format("Unknown error {}\n", err); }
  ```
  ```

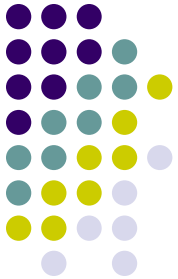
Variants: Basic use

- Think of a variant is like a tuple, but instead of holding all of its fields at once, it only contains one of them at a time
- Supports a very similar interface to tuples
- ```
  variant<int, double> v = 3; // Holds int
  get<0>(v); // Returns 3
  get<1>(v); // Throws std::bad_variant_access
  v = 3.5;   // Now holds a double
  get<double>(v); // Returns 3.5
  ```
- You can also check what is in it
  ```
  v.index; // returns 1
  holds_alternative<double>(v); // returns true
  holds_alternative<int>(v); // returns false
  ```
- We will learn some more ways to access variants when we cover object-oriented design
  ```

SOLID Principles

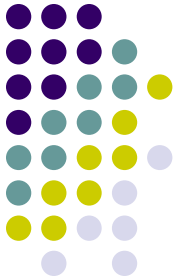
- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Create and extend classes with OO



- Full runtime-dispatch
- Must inherit
  - Very safe "nominal" typing
  - But can result in hard to maintain "spaghetti inheritance"
- Objects expensive to create with make_unique
- https://godbolt.org/z/T61ThMxzK

Create and extend classes with templates

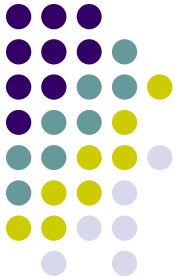


- Compile-time dispatch
- Can work with values instead of unique_ptrs
- Efficient
- Great open-closed support
- But cannot be used when runtime dispatch is needed
  - E.g., if the user chooses an animal type
- Or containers that need fixed types
  - `using Zoo = vector<???>;`
- https://godbolt.org/z/Pa4sqTeMo

Create and extend classes with duck-typed variants

- Supports runtime dispatch
- Can work with values instead of unique_ptrs
- (Usually) efficient
- Clunky notation
- https://godbolt.org/z/ec46Wc7fh

Type erasure



- Can be thought of as classes that act like templates
- This eliminates the "viralness" of templates, where you can get back to honest-to-goodness classes
- Used in the standard library by std::function and std::any
- With types erased, "Open-Closed" extension is difficult
- Our animal example
- https://godbolt.org/z/jGqEnaPoG

Static Polymorphism

- Used to be very common because people wanted to produce high performance code that "looks like" inheritance and overriding "OO"
  - "Traditional OO is better"
- Now less common
  - Lots of boilerplate
  - Disadvantages of templates (no runtime dispatch)
  - Disadvantages of OO (having to inherit is a headache)
- Still see it a lot in legacy code and when you want nominal typing together with static dispatch
- https://godbolt.org/z/be483Y3sh
- C++23's "Deducing this" feature will reduce the boilerplate, so it may become more popular