Casting

* Based on notes by Brad Lushman, used with ... I'll ask later.

Recall, in C:

```
Node n;
int *ip = (int *) &n;
```

Casting

* Based on notes by Brad Lushman, used with ... I'll ask later.

Recall, in C:

```
Node n;
int *ip = (int *) &n;
```

A **cast** forces the `Node *` to be treated as an `int *`.

In general, casts should be avoided.
More specifically, in C++, C-style casts should be avoided.

If you must use a C++ style cast, there are 4 kinds:

- `static_cast`
- `reinterpret_cast`
- `const_cast`
- `dynamic_cast`

static_cast

"Sensible casts" with well-defined semantics.

```
// double to int:
double d;
void f(int x);
void f(double x);
f(static_cast<int>(d)); // calls the int version of f
```

Note: decimal gets truncated. What if we want rounded instead?

static_cast

"Sensible casts" with well-defined semantics.

```
// double to int:
double d;
void f(int x);
void f(double x);
f(static_cast<int>(d)); // calls the int version of f
```

Note: decimal gets truncated. What if we want rounded instead?

```
// Superclass ptr to subclass ptr
Book *b = new Text{ ... };
Text *t = static_cast<Text *>(b);
```

You are taking responsibility that `b` actually points to a `Text`.

reinterpret_cast

Generally unsafe, implementation dependent, "weird" conversions.
Most uses result in undefined behaviour.

```
Student s;
Turtle *t = reinterpret_cast<Turtle *>(&s);
```

For when you want a `Student` to be treated as a `Turtle` - when is that??
Weird!

const_cast

For converting between const and non-const.
This is the only C++ cast that can *cast* away const-ness.

```
void g(int *p); // Knowing g won't actually modify *p

void f(const int *p) {
    ...
    g(const_cast<int *>(p));
    ...
}
```

dynamic_cast

Is it safe to convert a `Book *` to a `Text *`? Is this safe?

```
Book *pb = ...;
static_cast<Text *>(pb)->getTopic();
```

dynamic_cast

Is it safe to convert a `Book *` to a `Text *`? Is this safe?

```
Book *pb = ...;
static_cast<Text *>(pb)->getTopic();
```

Depends on what `pb` actually points at!

Better to try the cast first and see if it succeeds or not (a tentative cast).

```
Book *pb = ...;
Text *pt = dynamic_cast<Text *>(pb);
```

- If the cast works (`*pb` really is a `Text` or a subclass of `Text`), conversion is successful: `pt` will point at the object.
- If object is not the desired type, `pt` will be `nullptr` - you can then test for this.

```
if (pt) cout << pt->getTopic();
else cout << "Not a Text";
```

Casting and Smart Pointers

Previous examples used raw pointers but we can also cast smart pointers (`unique_ptr, shared_ptr`):

- `static_pointer_cast`
- `const_pointer_cast`
- `dynamic_pointer_cast`
- `reinterpret_pointer_cast`

Stay within the type: cast `shared_ptrs` to `shared_ptrs`.

Dynamic Casting with References

Yes you can do this too!

```
Text t{...};
Book &b = t;
Text &t2 = dynamic_cast<Text &>(b);
```

If `b` points to a `Text`, then `t2` is a reference to the same `Text`.

If not, then `t2` is `nullptr`?

Dynamic Casting with References

Yes you can do this too!

```
Text t{...};
Book &b = t;
Text &t2 = dynamic_cast<Text &>(b);
```

If `b` points to a `Text`, then `t2` is a reference to the same `Text`.

If not, then `t2` is `nullptr`?
No! There is no such thing as a *null reference*.
Raises an exception: `std::bad_cast`

Note: dynamic casting only works on classes with at least one virtual method - used for runtime type identification (RTTI).

Dynamic Reference Casting and the Polymorphic Assignment Problem

Dynamic reference casting offers a possible solution to the polymorphic assignment problem:

```
Text &Text::operator=(const Book &other) {  // virtual
    const Text &textother = dynamic_cast<const Text&>(other);
    // If other is not a Text then it throws

    if (this == &textother) return *this;
    Book::operator=(other);
    topic = textother.topic;
    return *this;
}
```

Is dynamic casting good style?

Good style?

You can use dynamic casting to make decisions based on an object's runtime type information (RTTI).

```
void whatIsIt(shared_ptr<Book> b) {
   if (dynamic_pointer_cast<Comic>(b))
      cout << "Comic";
   else if (dynamic_pointer_cast<Text>(b))
      cout << "Text";
   else if (b)
      cout << "Ordinary Book";
   else
      cout << "Nothing";
}
```

What would we say about the coupling of this with the `Book` hierarchy?

Good style?

You can use dynamic casting to make decisions based on an object's runtime type information (RTTI).

```
void whatIsIt(shared_ptr<Book> b) {
   if (dynamic_pointer_cast<Comic>(b))
      cout << "Comic";
   else if (dynamic_pointer_cast<Text>(b))
      cout << "Text";
   else if (b)
      cout << "Ordinary Book";
   else
      cout << "Nothing";
}
```

What would we say about the coupling of this with the `Book` hierarchy?
Highly coupled $\Rightarrow$ might indicate a bad design.
Why?

Bad Design?

Suppose you want to create a new type of `Book`, what changes would you need to make?

Bad Design?

Suppose you want to create a new type of `Book`, what changes would you need to make?

- must update `whatIsIt` to add a new clause.
- must find and fix all uses of dynamic casting before your code will work properly

$\Rightarrow$ easy to misuse, error prone

$\Rightarrow$ Better to use virtual methods!

Are all uses of dynamic casting indicative of bad design?

Good or Bad Design?

Are all uses of dynamic casting indicative of bad design?

No. `Text::operator=` (previous) does not require updates, etc - only needs to compare with it's own type (not everything in the hierarchy).

```
Text &Text::operator=(const Book &other) {  // virtual
    const Text &textother = dynamic_cast<const Text&>(other);
    // If other is not a Text then it throws

    if (this == &textother) return *this;
    Book::operator=(other);
    topic = textother.topic;
    return *this;
}
```

Fixing whatIsIt

Try to create an interface function that is uniform across all Book types.

```
class Book {
   ...
   virtual void identify() { cout << "Book"; }
};
...
void whatIsIt(Book *b) {
   if (b) b->identity();
   else cout << "Nothing";
}
```

Fixing whatIsIt

Try to create an interface function that is uniform across all Book types.

```
class Book {
   ...
   virtual void identify() { cout << "Book"; }
};
...
void whatIsIt(Book *b) {
   if (b) b->identity();
   else cout << "Nothing";
}
```

What if the interface isn't uniform across all types in the hierarchy?

Inheritance and virtual methods are well-suited when

- there is an unlimited number of specializations of a basic abstraction
- each follow the same interface

Adding a new subclass, for a new specialization, is easy.

Inheritance and virtual methods are well-suited when

- there is an unlimited number of specializations of a basic abstraction
- each follow the same interface

Adding a new subclass, for a new specialization, is easy.

BUT, what if we have the opposite case:

- there is a small number of specializations, all are known in advance and they are unlikely to change
- the different specializations may have very different interfaces

What do we need to do to add a new, unexpected, subclass?

Inheritance and virtual methods are well-suited when

- there is an unlimited number of specializations of a basic abstraction
- each follow the same interface

Adding a new subclass, for a new specialization, is easy.

BUT, what if we have the opposite case:

- there is a small number of specializations, all are known in advance and they are unlikely to change
- the different specializations may have very different interfaces

What do we need to do to add a new, unexpected, subclass?

- Must rework existing code to accommodate new interface.
  $\Rightarrow$ you weren't expecting to add a new subclass so you should expect to put in extra effort.

Example

```
class Turtle: public Enemy {
   void stealShell();
};

class Bullet: public Enemy {
   void deflect();
}
```

Interfaces are not uniform - a new enemy means a new interface
$\Rightarrow$ unavoidable work.

We could regard the set of enemy classes as fixed and maybe dynamic casting on enemies is justified.

Example

```
class Turtle: public Enemy {
   void stealShell();
};

class Bullet: public Enemy {
   void deflect();
}
```

Interfaces are not uniform - a new enemy means a new interface
$\Rightarrow$ unavoidable work.

We could regard the set of enemy classes as fixed and maybe dynamic casting on enemies is justified.

BUT, in this case, maybe inheritance isn't the correct abstraction mechanism to use.

Variant

If you know that an `Enemy` will only be a `Turtle` or a `Bullet` and you accept that adding new `Enemy` types will require widespread changes anyway, then consider:

```
import <variant>;
// An Enemy is either a Turtle or a Bullet
// old-style: typedef variant<...> Enemy;
using Enemy = variant<Turtle, Bullet>;


// Check what type e is:
if (holds_alternative<Turtle>(e) {
   cout << "Turtle";  // True if e is a Turtle
}
else ...
```

Variant

```
// Extracting the value:
try {
    Turtle t = get<Turtle>(e);
    // C++17 throws on error: bad_variant_access
    //use t ...
}
catch (std::bad_variant_access &) {
    // It wasn't a Turtle
}
```

A variant is like a union but it's *type-safe*.

- attempting to store as one type and fetch as another will throw an exception

Variant

If a variant is left uninitialized, what happens?

```
std::variant<Turtle, Bullet> e;
```

Variant

If a variant is left uninitialized, what happens?

```
std::variant<Turtle, Bullet> e;
```

The first option of the variant is default-constructed to initialize the variant.

What if the first option does not have a default constructor?

Variant

If a variant is left uninitialized, what happens?

```
std::variant<Turtle, Bullet> e;
```

The first option of the variant is default-constructed to initialize the variant.

What if the first option does not have a default constructor?
Compile error! (as we would expect)

Options:

Variant

If a variant is left uninitialized, what happens?

```
std::variant<Turtle, Bullet> e;
```

The first option of the variant is default-constructed to initialize the variant.

What if the first option does not have a default constructor?
Compile error! (as we would expect)

Options:

1. Make the first option a type that has a default ctor.
2. Don't define uninitialized variants.
3. Use `std::monostate` as the first option. This creates a "dummy" type that can be used as a default; i.e. can be used to create an "optional" type.

   ```
   variant<monostate, T> // = "T or nothing"
   ```

Also, `std::optional<T>` which contains a value or does not contain a value. Can convert to a Boolean T/F.