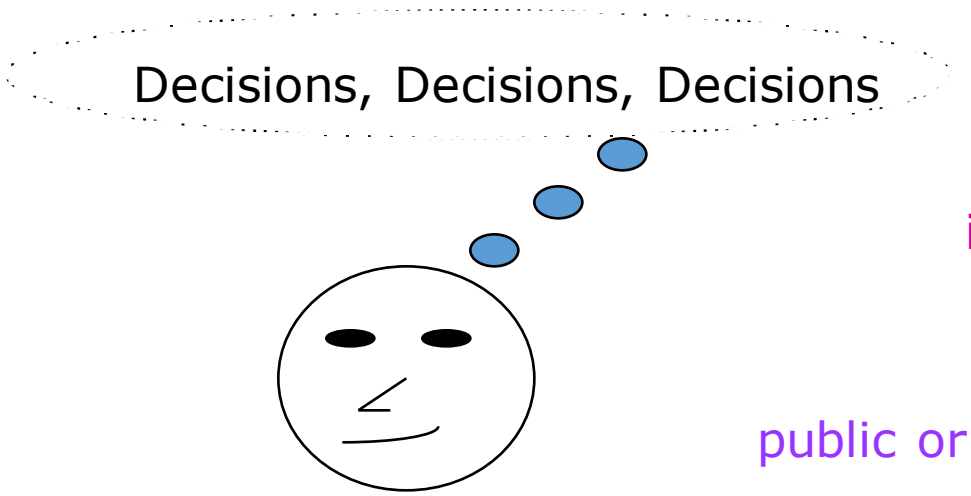
C++ can be really hard

"Used without discipline, however, C++ can lead to code that is incomprehensible, unmaintainable, inextensible, inefficient and just plain wrong."

What's it all about?



inheritance or templates?

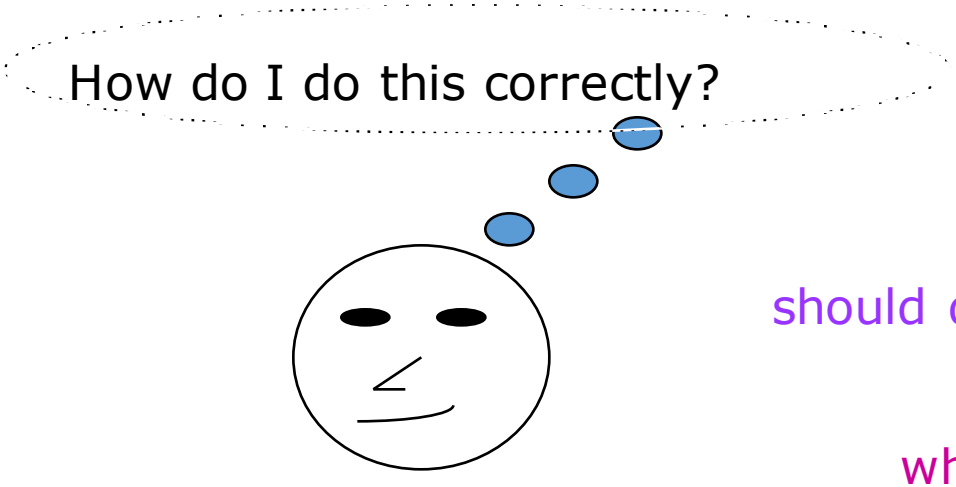
public or private inheritance?

private inheritance or composition?

member or non-member functions?

DESIGN PATTERNS!

What's it all about?



should destructor be virtual?

what return type to use?

what should operator do when it can't get enough memory?

Terminology

- **Declarations – name & type**

```
extern int x; // object declaration

// function declaration, also signature
std::size_t numDigits(int num);

class Widget;  // class declaration

// template declaration
template<typename T>
class GraphNode;
```

Terminology (continued)

- Definition – details

```
int x; // object definition – memory

// function definition - body
std::size_t numDigits(int num)
  { … }

// class and template definitions, include
// methods and data
class Widget{
   public:
     // list of methods, data…
};
template <typename T>
class GraphNode {
   public:
     // list of methods, data ..
};
```

Terminology (continued)

- Initialization – give first value
- Default constructor – no arguments
- Recommendation: make constructor explicit, to avoid implicit type conversions (example next slide)

Ex1: Explicit Example

```
class A{
public:
  A();
};
class B{
public:
  explicit B(int x=0,
 bool b=true);
};

void doSomething(B
 bObj);
```

```
// in main…
B bObj1;
doSomething(bObj1); // fine

B bObj2(28);
doSomething(bObj2); // fine

doSomething(28); // error!
doSomething(B(28));
// fine, uses B constructor
// explicitly
```

Ex2: Copy Constructor/Copy Assignment

```
class Widget {
public:
  Widget();
  Widget(const Widget& rhs);
  Widget& operator =(const Widget& rhs);
 …
};
Widget w1;
// invoke default constructor
Widget w2(w1);
// invoke copy constructor
w1 = w2;
// invoke assignment
Widget w3 = w2;
// invoke copy constructor!
```

Ex3

```
#define PI 3.14;
const double pi = 3.14;
```

- Symbolic names may be removed, don't show up in error messages or debugging – confusing.
- Could have multiple copies of 3.14 in object code.

Prefer consts, enums and inlines to #defines (i.e., prefer compiler to preprocessor)

More on constants

```
class CostEstimate {
private:
  static const int NumTurns=5;
  static const double FudgeFactor;
};

const double CostEstimate::FudgeFactor = 1.35;
```

remember static?

some compilers won't allow values in "declaration" – must provide definition

enum "hack" – if need value for constant and compiler won't allow

```
class GamePlayer {
private:
  enum {NumTurns = 5; }
  int scores[NumTurns];
};
```

Ex4

Use const wherever possible

```
char greeting[] = “Hello”;

char *p = greeting;
// non-const pointer, non-const data

const char *p2 = greeting;
// non-const pointer, const char data

char * const p3 = greeting;
// const pointer, non-const data

const char * const p4 = greeting;
// const pointer, const data
```

Placement of const

- `void f1 (const Widget *pw);`
  `void f2 (Widget const *pw);`
  They are equivalent!

- Iterators are similar to pointers:
  - a const iterator is like T *const … iterator can't point to a different item, but you can change the value of the item that it is pointing to
  - a const_iterator is like const T*… iterator can point to a different item, but item can't be changed

Consider making return value const

```
const Rational { . . .
const Rational operator *(const Rational& lhs,
  const Rational& rhs);
};
Rational a, b, c;
if (a * b = c) . . .
```

Meant to be ==

What will happen with const? without?

What happens with built-in types?

May need two versions

- Can't overload based on return type, but can overload based on const vs. non-const member function

```
class TextBlock {
public:
. . .
const char& operator[](std::size_t pos) const
  { return text[pos]; }
char& operator[](std::size_t pos)
  { return text[pos]; }
private:
   std::string text;
};
void print(const TextBlock& ctb)
{
     std::cout << ctb[0];
// OK
     ctb[0] = 'A';
// Not OK – compiler error
}
TextBlock tb("hello");
tb[0] = 'H';
// OK because return has &, not const
```

Physical constness vs Logical constness

- Physical (bitwise) const: member function is const iff it doesn't modify any of the bits inside the object
- Logical const: const member method might modify some bits in object, but only in ways clients cannot detect
- Compilers enforce bitwise constness, you should program using logical constness

Example

```
class CTextBlock {
public:
    …
    char& operator[](std::size_t pos) const
      {return pText[pos]; }
private:
  char *pText;
};

const CTextBlock cctb("Hello");
// constant object
char *pc = &cctb[0];
// calls constant [] operator
*pc = 'J';
// cctb is now "Jello"
```

violates logical constness, but compiler allows!

Modifying bits client doesn't see

```
class CTextBlock {
public:
 . . .
 std::size_t length() const;
private:
  char *pText;
  std::size_t textLength; // last calculated length
  bool lengthIsValid; // whether length is valid
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid)
  {
    // error! changes bits
    textLength = std::strlen(pText);
    lengthIsValid = true;
  }
  return textLength;
}
```

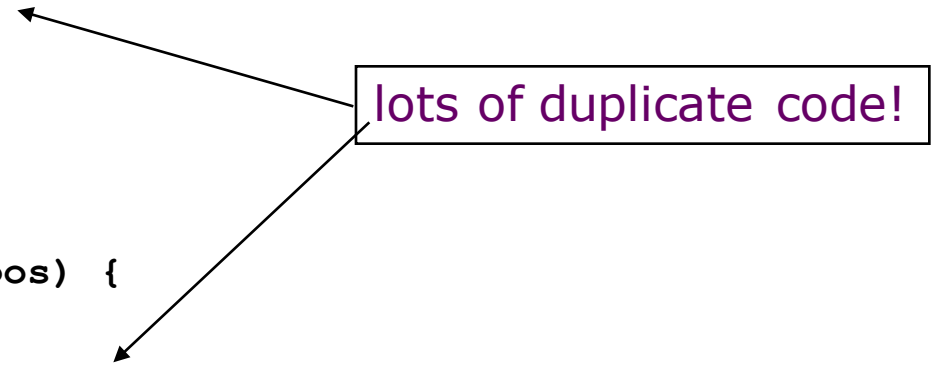
Mutable to the rescue

```
class CTextBlock {
public:
 . . .
 std::size_t length() const;
private:
  char *pText;
  mutable std::size_t textLength;
  mutable bool lengthIsValid;};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText); // OK now
   lengthIsValid = true;
  }
  return textLength;
}
```

Avoid duplication in const/non-const

```
class TextBook {
public:
…
const char& operator[](std::size_t pos) const {
… // do bound checking
… // log access data
… // verify data integrity
return text[pos];
}
char& operator[](std::size_t pos) {
… // do bound checking
… // log access data
… // verify data integrity
return text[pos];
}
private:
std::string text;
};
```



could put duplicated code in a function and call it – but then have duplicated calls to that function, and duplicated return

Cast-Away const

```
class TextBook {
public:
…
const char& operator[](std::size_t pos) const {
… // same as before
return text[pos];
}

char& operator[](std::size_t pos) {
return const_cast<char&>(
    static_cast<const TextBlock&>(*this)[position]); }

private:
std::string text;
};
```

Now non-const [] just calls const

const_cast needed to **remove** const before return

add const, to call const version of []
safe conversion, so use static_cast

DO NOT go the other direction – not safe!

Ex4

Make sure that objects are initialized before they're used.

always initialize objects before use.

Initialization

- Make sure all constructors initialize everything in the object.
- Assignment is not the same as initialization.

```
ABEntry::ABEntry(const std::string&name,
 const std::list<PhoneNumber>& phones)
{
    theName = name;
    thePhones = phones;
    numTimesConsulted = 0;
}
```

default constructors were called for these prior to entering the body of the constructor – that's when they were initialized. Not true for built-in types (e.g., numTimesCalled).

Initialization (continued)

- Prefer member initialization lists:

```
ABEntry::ABEntry(const string& name,
 const list<PhoneNumber>& phones) :
 theName(name), thePhones(phones),
 numTimesConsulted (0) {}
```

- Single call to copy constructor is more efficient than call to default constructor followed by call to copy assignment.
- No difference in efficiency for numTimesConsulted, but put in list for consistency

Initialization (continued)

- Can do member initialization lists even for default construction:

```
ABEntry::ABEntry() : theName(),
 thePhones(), numTimesConsulted (0)
 {}
```

- Members are initialized in the order they are listed in class. Best to list them in that order in initialization list.
- Base classes are always initialized before subclasses.

Initialization of non-local static objects

```
class FileSystem {
public:
   std::size_t numDisks() const;
  . . .
};
extern FileSystem tfs; // declaration, must be defined
                       // in some .cpp in your library

class Directory{
public Directory(params);
};
Directory::Directory(params) {
…
std::size_t disks = tfs.numDisks(); // use tfs object
}
```

Has tfs been initialized?

Initialization of non-local (continued)

```
class FileSystem { … } // as before

FileSystem& tfs() {
    static FileSystem fs;
    return fs;
}
class Directory{ … } // as before

Directory::Directory(params) {
…
std::size_t disks = tfs().numDisks();
// calls tfs function now
}
```

SINGLETON DESIGN PATTERN

Ex5

Know what functions C++ silently writes and calls.

```
class Empty{};
```

becomes:

```
class Empty{
public:
Empty() { ... }
Empty(const Empty& rhs) { … }
~Empty() { … }
Empty& operator=(const Empty& rhs) {…}
};
```

What do they do?

- Copy constructor and assignment generally do a field-by-field copy.
- These functions will not be written if your class includes a const value or a reference value (compiler isn't sure how to handle).

```
template <typename T>
class NamedObject {
public:
NamedObject(std::string& name, const T&
 value);
private:
   std::string& nameValue;
   const T objectValue;
};
```

Ex6

Explicitly disallow the use of compiler-generated functions you do not want

- By declaring member functions explicitly, you prevent compilers from generating their own version.
- By making a function private, you prevent other people from calling it. – don't define them, so anyone who tries will get a linker error
- Even better, put functions in parent class, if child class attempts to call will generate a compiler error (earlier detection is better).

Example

```
class Uncopyable {
protected:
  Uncopyable();
  ~Uncopyable();
private:
  Uncopyable(const Uncopyable&);
  Uncopyable& operator=(const Uncopyable&);
};

class HomeForSale: private Uncopyable {
  … // class has no copy ctor or = operator
};
```

Ex7

Declare destructors virtual in polymorphic base classes.

```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
 …
};
class AtomicClock : public TimeKeeper { … };
class WristWatch : public TimeKeeper { … };
TimeKeeper* getTimeKeeper();
// returns pointer to dynamically allocated object
TimeKeeper *ptk = getTimeKeeper(); // AtomicClock
… // use it in some way
delete ptk; // release it
```

THE RESULTS OF THIS OPERATION ARE UNDEFINED!

TimeKeeper continued

- Most likely AtomicClock part of object would not be destroyed – a “partially destroyed” object
- Solution:

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
 …
};
```

- Any class with virtual functions should almost certainly have a virtual destructor.

Don’t always make it virtual…

- If a class does not contain any virtual functions, often indicates its not intended to be a base class. Making destructor virtual would be a bad idea.

```
class Point {
public:
  Point(int xCoord, int yCoord);
  ~Point();
private:
  int x, y;
};
```

- Point class can fit in 64-bit register, be passed as 64-bit value to other languages (C/Fortran)
- Virtual functions require objects to carry extra info for runtime binding. Typically a vptr (virtual table pointer) that points to an array of function pointers called a vtbl (virtual table).
- Point class will now be 96 bits (on 32-bit architecture)

Handy rule: declare a virtual destructor if and only if that class contains at least one other virtual function.

When not to extend…

- Be careful when you choose to extend a class. std::string contains no virtual functions, so is not a good choice for a base class. STL container types also do not have virtual destructors.

```
class SpecialString : public std::string
 { … };
SpecialString *pss = new SpecialString("Doomed");
std::string *ps;
…
ps = pss;
….
delete ps;  // UNDEFINED
```

Java can prevent programmers from extending a class… C++ can't

A destructor trick

- Maybe you have a class that you want to be abstract, but you don't have any pure virtual functions.
- Make the destructor pure virtual
- BUT – you still have to provide a definition, because the compiler always calls the base class destructor.

```
class AWOV {
public:
virtual ~AWOV()=0;
};
AWOV::~AWOV()  {}
```

But is it really polymorphic?

- The “handy rule” for base classes really applies only to *polymorphic* base classes – those designed to allow manipulation of derived class objects through base class interfaces.
- STL, string
- Not always the case – Uncopyable, for example, is designed to prevent copying. You wouldn’t do:

```
Uncopyable *uc;
uc = new HomeForSale();
```

Ex8

Prevent exceptions from leaving destructors

- Why? What if you had ten Widgets in an array and the ~Widget for the first Widget threw an exception. Then ~Widget is invoked for the next Widget in the array, and it throws an exception.

Example: DBConnection

```
class DBConnection{
public:
 … // params omitted for simplicity
  static DBConnection create();

  void close(); // may throw exception
};

class DBConn { //manages DBConnection
public:
  …
// destructor ensures db connection always closed
  ~DBConn() { db.close(); }
private:
};
```

DBConnection (continued)

- Allows clients to:

```
{
DBConn.dbc(DBConnection::create());
… // use object
} // destructor called at end of block
```

- Problem: if db.close() fails, exception will be thrown in destructor

Options for destructor

- Terminate the program (OK if program cannot continue to run after this type of error)

```
DBConn::~DBConn()  {
try { db.close() }
catch(…) {
  make log entry that call failed
  std::abort();
}
```

- Swallow the exception (usually a bad idea)

```
DBConn::~DBConn()  {
try { db.close() }
catch(…) {
  make log entry that call failed
}
```

A better approach…

```
class DBConn { //manages DBConnection
public:
  void close() { // gives client option
    db.close();
    closed = true;
  }

~DBConn() {
   if (!closed) {
   try {db.close();} // backup in case client didn't
   catch ( . . ) {
     make log entry that call failed
     … // terminate or swallow
   } } }
private:
   DBConnection db;
   bool closed;
};
```