Where do we go from here?

- **C++**
  - classes and objects, with all the moving parts visible
  - operator overloading
  - templates, STL, standards, …

- **Java**
  - components, collections, generics
  - language and performance comparisons

- **Go**
  - concurrency

- **Little languages**
  - languages you might build yourself

- ???
Program structure issues

- how to cope with ever bigger programs?
- objects
  - user-defined data types
- components
  - related objects
- frameworks
  - automatic generation of routine code
- interfaces
  - boundaries between code that provides a service and code that uses it
- information hiding
  - what parts of an implementation are visible
- resource management
  - creation and initialization of entities
  - maintaining state
  - ownership: sharing and copying
  - memory management
  - cleanup
- error handling; exceptions
C++

- **designed & implemented by Bjarne Stroustrup**
  - started ~ 1980; ISO C++98 standard; C++11; C++14; C++17 nearly ready
- **a better C**
  - almost completely upwards compatible with C
  - more checking of interfaces (e.g., function prototypes, added to ANSI C)
  - other features for easier programming
- **data abstraction**
  - methods reveal only WHAT is done
  - classes hide HOW something is done in a program, can be changed as program evolves
- **object-oriented programming**
  - *inheritance* -- define new types that inherit properties from previous types
  - *polymorphism* or dynamic binding -- function to be called is determined by data type of specific object at run time
- **templates or "generic" programming**
  - compile-time parameterized types
  - define families of related types, where the type is a parameter
- **a "multi-paradigm" language**
  - lots of ways to write code
Did you really say that?

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

– What people tend to miss, is that what I said there about C++ is to a varying extent true for all powerful languages. As you protect people from simple dangers, they get themselves into new and less obvious problems.

"Within C++, there is a much smaller and cleaner language struggling to get out." [...] And no, that smaller and cleaner language is not Java or C#.

“There are more useful systems developed in languages deemed awful than in languages praised for being beautiful -- many more”
C++ synopsis

• **data abstraction with classes**
  – a class defines a type that can be used to declare variables of that type, control access to representation

• **operator and function name overloading**
  – almost all C operators (including =, +=..., ( ), [ ], ->, argument passing and function return) can be overloaded to apply to user-defined types

• **control of creation and destruction of objects**
  – initialization of class objects, recovery of resources on destruction

• **inheritance: derived classes built on base classes**
  – virtual functions override base functions
  – multiple inheritance: inherit from more than one class

• **exception handling**

• **namespaces for separate libraries**

• **templates (generic types)**
  – Standard Template Library: generic algorithms on generic containers
  – template metaprogramming: execution of C++ code *during compilation*

• **(almost) upward compatible with C except for new keywords**
Stack class in C++

// stk1.c: simple-minded stack class
class stack {
  private: // default visibility
    int stk[100];
    int *sp;
  public:
    int push(int);
    int pop();
    stack(); // constructor decl
};

int stack::push(int n) {
  return *sp++ = n;
}
int stack::pop() {
  return *--sp;
}
stack::stack() { // constructor implementation
  sp = stk;
}

stack s1, s2; // calls constructors
s1.push(1);   // calls method
s2.push(s1.pop());
Inline definitions

• member function body can be written inside the class definition
• this normally causes it to be implemented inline
  – no function call overhead

// stk2.c: inline member functions

class stack {
    int stk[100];
    int *sp;
    public:
        int push(int n) { return *sp++ = n; }
        int pop() { return *--sp; }
        stack() { sp = stk; }
};
Memory allocation: **new** and **delete**

- **new** is a type-safe alternative to malloc
  - **delete** is the matching alternative to free
- **new** $T$ allocates an object of type $T$, returns pointer to it
  
  ```
  stack *sp = new stack;
  ```
- **new** $T[n]$ allocates array of $T$'s, returns pointer to first
  
  ```
  int *stk = new int[100];
  ```
  - by default, throws exception if no memory
- **delete** $p$ frees the single item pointed to by $p$ 
  
  ```
  delete sp;
  ```
- **delete** $[\ ]$ $p$ frees the array beginning at $p$
  
  ```
  delete [] stk;
  ```
- **new** uses $T$'s constructor for objects of type $T$
  - need a default constructor for array allocation
- **delete** uses $T$'s destructor $\sim T()$
- **use** **new**/**delete** instead of **malloc**/**free**
  - **malloc**/**free** provide raw memory but no semantics 
  - this is inadequate for objects with state
  - **never** mix **new**/**delete** and **malloc**/**free**
Dynamic stack with **new**, **delete**

`// stk3.c: new, destructors, delete`

class stack {
    private:
        int *stk;    // allocated dynamically
        int *sp;     // next free place
    public:
        int push(int);
        int pop();
        stack();    // constructor
        stack(int n); // constructor
        ~stack();   // destructor
    };

stack::stack() {
    stk = new int[100];  sp = stk;
}
stack::stack(int n) {
    stk = new int[n];  sp = stk;
}
stack::~stack() {
    delete [ ] stk;
}
Constructors and destructors

• **constructor:**
  creating a new object (including initialization)
  – implicitly, by entering the scope where it is declared
  – explicitly, by calling `new`

• **destructor:**
  destroying an existing object (including cleanup)
  – implicitly, by leaving the scope where it is declared
  – explicitly, by calling `delete` on an object created by `new`

• **construction includes initialization, so it may be parameterized**
  – by multiple constructor functions with different args
  – an example of function overloading

• **new can be used to create an array of objects**
  – in which case `delete` can delete the entire array
Implicit and explicit allocation and deallocation

• implicit:

```cpp
f() {
    int i;
    stack s;    // calls constructor stack::stack()
    ...  
    // calls s::~stack() implicitly
}
```

• explicit:

```cpp
f() {
    int *ip = new int;
    stack *sp = new stack;    // calls stack::stack()
    ...
    delete sp;    // calls sp->~stack()
    delete ip;
    ...
}
```
Constructors; overloaded functions

- two or more functions can have the same name if the number and/or types of arguments are different

```cpp
abs(int); abs(double); abs(complex)
atan(double x); atan(double y, double x);

int abs(int x) { return x >= 0 ? x : -x; }
double abs(double x) { return x >= 0 ? x : -x; }
```

- multiple constructors for a class are a common instance

```cpp
stack::stack();
stack::stack(int stacksize);

stack s; // default stack::stack()
stack s1(); // same
stack s2(100); // stack::stack(100)
stack s3 = 100; // also stack::stack(100)
```
Operator overloading

- **almost all C operators can be overloaded**
  - a new meaning can be defined when one operand of an operator is a user-defined (class) type
  - define *operator* + for object of type T
    
    ```
    T T::operator+(int n) {...}
    T T::operator+(double d) {...}
    ```
  - define regular + for object(s) of type T
    
    ```
    T operator +(T f, int n) {...}
    ```
  - can't redefine operators for built-in types
    
    ```
    int operator +(int, int) is ILLEGAL
    ```
  - can't define new operators
  - can't change precedence and associativity
    
    e.g., ^ is low precedence even if used for exponentiation

- **3 short examples**
  - complex numbers: overloading arithmetic operators
  - IO streams: overloading << and >> for input and output
  - subscripting: overloading [ ]

- **later:** overloading assignment and function calls
An implementation of complex class

class complex {
    double re, im;
    public:
        complex(double r = 0, double i = 0)
        { re = r; im = i; }  // constructor

        friend complex operator +(complex,complex);
        friend complex operator *(complex,complex);
    }

complex operator +(complex c1, complex c2) {
    return complex(c1.re+c2.re, c1.im+c2.im);
}

• complex declarations and expressions

    complex a(1.1, 2.2), b(3.3), c(4), d;

    d = 2 * a;
    2 coerced to 2.0 (C promotion rule)
        then constructor invoked to make complex(2.0, 0.0)

• operator overloading works well for arithmetic types
References: controlled pointers

• need a way to access object, not a copy of it
• in C, use pointers
  
  ```c
  void swap(int *x, int *y) {
    int temp;
    temp = *x; *x = *y; *y = temp;
  }
  swap(&a, &b);
  ```

• in C++, references attach a name to an object
• a way to get "call by reference" (var) parameters without using explicit pointers
  
  ```c
  void swap(int &x, int &y) {
    int temp;
    temp = x; x = y; y = temp;
  }
  swap(a, b);  // pointers are implicit
  ```

• because it's really a pointer, a reference provides a way to access an object without copying it
A vector class: overloading [ ]

class ivec {  // vector of ints
    int *v;         // pointer to an array
    int size;       // number of elements
public:
    ivec(int n) { v = new int[size = n]; }

    int& operator[](int n) {  // checked
        assert(n >= 0 && n < size);
        return v[n];
    }
};

ivec iv(10);   // declaration
iv[10] = 1;    // checked access on left side of =

• operator[ ] returns a reference
• a reference gives access to the object so it can be changed
• necessary so we can use [ ] on left side of assignment
Input and output with iostreams

- overload operator `<<` for output and `>>` for input
  - very low precedence, left-associative, so
    
    ```
    cout << e1 << e2 << e3
    ```
  - is parsed as
    
    ```
    (((cout << e1) << e2) << e3)
    ```

```cpp
#include <iostream>

ostream& operator<<(ostream& os, const complex& c) {
    os << "(" << c.real() << " , " << c.imag() << ")";
    return os;
}

while (cin >> name >> val) {
    cout << name << " = "
    << val << "\n";
}
```

- takes a reference to iostream and data item
- returns the reference so can use same iostream for next expression
- each item is converted into the proper type
- iostreams `cin`, `cout`, `cerr` already open (== `stdin`, `stdout`, `stderr`)
Formatter in C++

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

const int maxlen = 60;
string line;
void addword(const string&);
void printline();

main(int argc, char **argv) {
    string word;
    while (cin >> word)
        addword(word);
    printline();
}
void addword(const string& w) {
    if (line.length() + w.length() > maxlen)
        printline();
    if (line.length() > 0)
        line += " ";
    line += w;
}
void printline() {
    if (line.length() > 0) {
        cout << line << endl;
        line = "";
    }
}
```
Life cycle of an object

• **construction: creating a new object**
  – implicitly, by entering the scope where it is declared
  – explicitly, by calling `new`
  – construction includes initialization

• **copying: using existing object to make a new one**
  – "copy constructor" makes a new object from existing one of the same kind
  – implicitly invoked in (some) declarations, function arguments, function return

• **assignment: changing an existing object**
  – occurs explicitly with `=, +=`, etc.
  – meaning of explicit and implicit copying must be part of the representation
    default is member-wise assignment and initialization

• **destruction: destroying an existing object**
  – implicitly, by leaving the scope where it is declared
  – explicitly, by calling `delete` on an object created by `new`
  – includes cleanup and resource recovery
Strings: constructors & assignment

• another type that C and C++ don't provide
• implementation of a String class combines
  – constructors, destructors, copy constructor
  – assignment, operator =
  – constant references
  – handles, reference counts, garbage collection

• Strings should behave like strings in Awk, Python, Java, ... 
  – can assign to a string, copy a string, etc.
  – can pass them to functions, return as results, ...

• storage managed automatically
  – no explicit allocation or deletion
  – grow and shrink automatically
  – efficient

• can create String from "..." C char* string
• can pass String to functions expecting char*
"Copy constructor"

- when a class object is passed to a function, returned from a function, or used as an initializer in a declaration, a copy is made:
  
  ```
  String substr(String s, int start, int len)
  ```

- a "copy constructor" creates an object of class X from an existing object of class X

- obvious way to write it causes an infinite loop:
  
  ```
  class String {
      String(String s) {...} // doesn't work
  };
  ```

- copy constructor parameter must be a reference so object can be accessed without copying
  
  ```
  class String {
      String(const String& s) {...}
      // ...
  };
  ```

- copy constructor is necessary for declarations, function arguments, function return values
String class

class String {
    private:
        char    *sp;
    public:
        String() { sp=strdup(""); }  // String s;
        String(const char *t) { sp=strdup(t); } // String s("abc");
        String(const String &t) { sp=strdup(t.sp); } // String s(t);
        ~String() { delete [] sp; }

        String& operator =(const char *);// s="abc"
        String& operator =(const String &);// s1=s2

        const char *s() { return sp; } // as char*
    };

• assignment is not the same as initialization
    – changes the state of an existing object

• the meaning of assignment is defined by a member function named operator=
    x = y means x.operator=(y)
Assignment operators

```cpp
String& String::operator =(const char *t) { // s = "abc"
    delete [] sp;
    sp = strdup(t);
    return *this;
}
String& String::operator=(const String& t) { // s1 = s2
    if (this != &t) {  // avoid s1 = s1
        delete [] sp;
        sp = strdup(t.sp);
    }
    return *this;
}
```

- in a member function, `this` points to current object, so `*this` is the object (returned as a reference)
- assignment operators almost always end with
  ```cpp
  return *this;
  ```
  which returns a reference to the LHS
  - permits multiple assignment `s1 = s2 = s3`
Handles and reference counts

- how to avoid unnecessary copying for classes like strings, arrays, other containers

- copy constructor may allocate new memory even if unnecessary
  - e.g., in `f(const String& s)` string value would be copied even if it won't be changed by `f`

- a handle class manages a pointer to the real data
- implementation class manages the real data
  - string data itself
  - counter of how many Strings refer to that data
  - when String is copied, increment the ref count
  - when String is destroyed, decrement the ref count
  - when last reference is gone, free all allocated memory

- with a handle class, copying only increments reference count
  - "shallow" copy instead of "deep" copy
Reference counts

\[
s = "abc"
\]

\[
t = s
\]

\[
t = "def"
\]
Reference counts or garbage collection?

"Reference counting, for instance, is occasionally useful but is more often a sign that someone isn't thinking carefully about ownership."

Google C++ style guide

"I consider garbage collection a last choice and an imperfect way of handling resource management. That does not mean that it is never useful, just that there are better approaches in many situations."

Bjarne Stroustrup
Inheritance

• a way to create or describe one class in terms of another
  – "a D is like a B, with these extra properties..."
  – "a D is a B, plus"
  – B is the base class or superclass
  – D is the derived class or subclass
    C++, Perl, Python, ... use base/derived; Java, Ruby, ... use super/sub

• inheritance is used for classes that model strongly related concepts
  – objects share some common properties, behaviors, ...
  – and have some properties and behaviors that are different

• base class contains aspects common to all
• derived classes contain aspects different for different kinds
Derived classes

```cpp
class Shape {
    int color;
    virtual Shape& draw();
    // other items common to all Shapes
};
class Rect: public Shape {
    Point origin; double ht, wid;
    Shape& draw() {...} // how to draw a rectangle
};
class Circle: public Shape {
    Point center; double rad;
    Shape& draw() {...} // how to draw a circle
};
```

• a Rect is a derived class of (a kind of) Shape
  – a Rect "is a" Shape
  – inherits all members of Shape
  – adds its own members

• a Circle is also a derived class of Shape
  – adds its own different members
Virtual Functions

• a function in a base class that can be overridden by a function in a derived class (with same name and arguments)

    class Shape {
        public:
            virtual Shape& draw();
            ...
    };

    "virtual" means that a derived class may provide its own version of this function, which will be called automatically for instances of that derived class

• the base class can provide a default implementation

• if the base class is "pure", it must be derived from
    – pure base class can't exist on its own; no default implementation
Polymorphism

- when a pointer or reference to a base-class type points to a derived-class object
- and you use that pointer or reference to call a virtual function
- this calls the derived-class function
- "polymorphism": proper function to call is determined at run-time
- e.g., drawing Shapes on a linked list:

```c
draw_all(Shape *sp) {
    for ( ; sp != NULL; sp = sp->next)
        sp->draw();
}
```

- the virtual function mechanism automatically calls the right draw() function for each object
- the loop does not change if more kinds of Shapes are added
Implementation of virtual functions

• each class object that has virtual functions has one extra word that holds a pointer to a table of virtual function pointers ("vtbl")
• each class with virtual functions has one vtbl
• a call to a virtual function calls it indirectly through the vtbl
Summary of inheritance

- a way to describe a family of types
- by collecting similarities (base class)
- and separating differences (derived classes)

- polymorphism: proper member functions determined at run time
  - virtual functions are the C++ mechanism

- not every class needs inheritance
  - may complicate without compensating benefit

- use composition instead of inheritance?
  - an object contains an (has) an object rather than inheriting from it

- "is-a" versus "has-a"
  - inheritance describes "is-a" relationships
  - composition describes "has-a" relationships
Templates (parameterized types, generics)

- another approach to polymorphism
- compile time, not run time
- a template specifies a class or a function that is the same for several types
  - except for one or more type parameters

- e.g., a vector template defines a class of vectors that can be instantiated for any particular type
  
  ```
  vector<int>
  vector<String>
  vector<vector<int>>
  ```

- templates versus inheritance:
  - use inheritance when behaviors are different for different types
    drawing different Shapes is different
  - use template when behaviors are the same, regardless of types
    accessing the n-th element of a vector is the same, no matter what type the vector is
Vector template class

• vector class defined as a template, to be instantiated with different types of elements

```cpp
template <typename T> class vector {
    T *v;     // pointer to array
    int size; // number of elements

gpublic:
    vector(int n=1) { v = new T[size = n]; }
    T& operator [](int n) {
        assert(n >= 0 && n < size);
        return v[n];
    }
};
```

```
vector<int> iv(100);       // vector of ints
vector<complex> cv(20);    // vector of complex
vector<vector<int> > vvi(10); // vector of vector of int
vector<double> d;          // default size
```

• compiler instantiates whatever types are used
Standard Template Library (STL)

Alex Stepanov

(GE > Bell Labs > HP > SGI > Compaq > Adobe > A9 > ...)

• general-purpose library of containers (vector, list, set, map, …)
  generic algorithms (find, replace, sort, …)
• algorithms written in terms of iterators performing specified access patterns on containers
  – rules for how iterators work, how containers have to support them

• generic: every algorithm works on a variety of containers, including built-in types
  – e.g., find elements in char array, vector<int>, list<…>

• iterators: generalization of pointer for uniform access to items in a container
Containers and algorithms

• **STL container classes contain objects of any type**
  – sequences: vector, list, slist, deque
  – sorted associative: set, map, multiset, multimap
    hash_set and hash_map are in C++11, as "unordered_set" and "unordered_map"

• **each container class is a template that can be instantiated to contain any type of object**

• **generic algorithms**
  – find, find_if, find_first_of, search, ...
  – count, min, max, ...
  – copy, replace, fill, remove, reverse, ...
  – accumulate, inner_product, partial_sum, ...
  – sort
  – binary_search, merge, set_union, ...

• **performance guarantees**
  – each combination of algorithm and iterator type specifies worst-case (O(…)) performance bound
    e.g., maps are O(log n) access, vectors are O(1) access
Iterators

• a generalization of C pointers
  
  for (p = begin; p < end; ++p)
  
  do something with *p

• range from begin() to just before end()  [begin, end)

• ++iter advances to the next if there is one

• *iter dereferences (points to value)

• uses operator != to test for end of range
  
  for (iter i = v.begin(); i != v.end(); ++i)
  
  do something with *i

#include <vector>
#include <iterator>
using namespace ::std;

int main() {

    vector<double> v;
    for (int i = 1; i <= 10; i++)
        v.push_back(i);

    vector<double>::const_iterator it;
    double sum = 0;
    for (it = v.begin(); it != v.end(); ++it)
        sum += *it;

    printf("%g\n", sum);
}

Iterators (2)

- no change to loop if type or representation changes

- not all containers support all iterator operations

- input iterator
  - can only read items in order, can't store into them (e.g., input from file)

- output iterator
  - can only write items in order, can't read them (output to a file)

- forward iterator
  - can read/write items in order, can't go backwards (singly-linked list)

- bidirectional iterator
  - can read/write items in either order (doubly-linked list)

- random access iterator
  - can access items in any order (array)
Example: STL sort

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() { // sort stdin by lines
    vector<string> vs;
    string tmp;
    while (getline(cin, tmp))
        vs.push_back(tmp);
    sort(vs.begin(), vs.end());
    copy(vs.begin(), vs.end(),
         ostream_iterator<string>(cout, "\n");
}

• vs.push_back(s) pushes s onto "back" (end) of vs
• 3rd argument of copy is a "function object" that calls a function for each iteration
    – uses overloaded operator()
```
Template metaprogramming

- do computation at compile time to avoid computation at run time
  - evaluating constants, unrolling loops, building data structures

// from Effective C++ 3e, by Scott Meyers

#include <iostream>
using namespace ::std;

template<unsigned n> struct Factorial {
    enum { value = n * Factorial<n-1>::value }
};
template<> struct Factorial<0> {
    enum { value = 1 }
};

int main() {
    std::cout << Factorial<5>::value << "\n";
    std::cout << Factorial<10>::value << "\n";
}
What to use, what not to use?

• **Use**
  - classes
  - const
  - const references
  - default constructors
  - C++-style casts
  - bool
  - new / delete
  - C++ string type
  - range for
  - auto

• **Use sparingly / cautiously**
  - overloaded functions
  - inheritance
  - virtual functions
  - exceptions
  - STL

• **Don't use**
  - malloc / free
  - multiple inheritance
  - run time type identification
  - references if not const
  - overloaded operators (except for arithmetic types)
  - default arguments (overload functions instead)