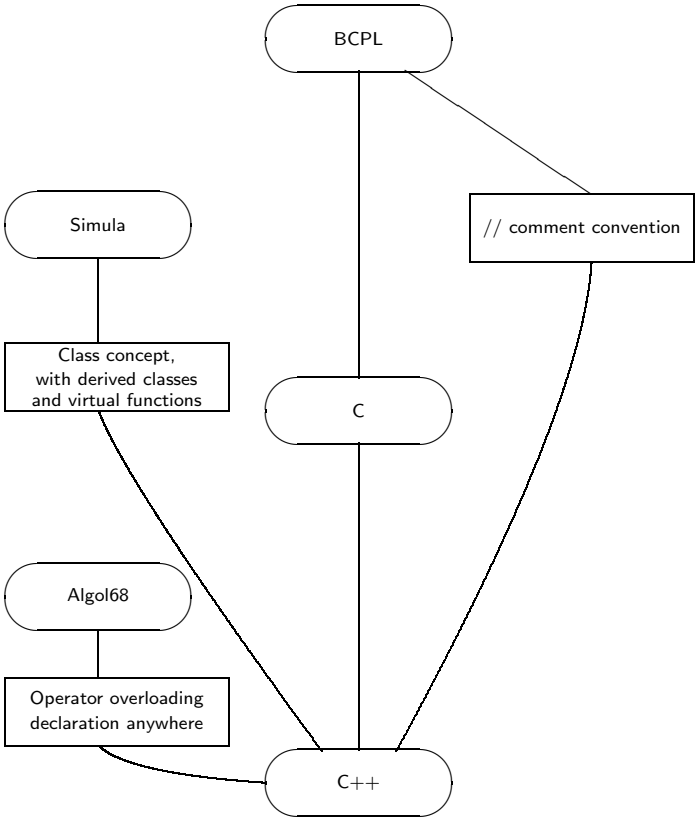
Introduction to C++

History of C++

- First released in 1983
- Derived from C and retains C as a subset
- C itself was inspired by BCPL
  - The // convention of BCPL is brought back into C++
- Another inspiration came from Simula67
  - Simula67 was the first object-oriented programming language, designed to solve simulation problems
  - Class concept, with derived classes and virtual functions
- Algol68
  - Operator overloading
  - Facility to place a declaration anywhere in the code



- Goals
  - Simplicity
  - Compatibility with C, with some cleaning up of C syntax
  - Make writing *good* programs easier and enjoyable
  - Extensions

    - Multiple inheritance
    - `static` member functions and `const` member functions
    - `protected` members
    - Information hiding
        - All members in a C `struct` are visible to the programmer, and can be so manipulated
        - C++ gets around this by letting the person who designed the structure determine the members that can be viewed or manipulated
    - Polymorphism (virtual functions)
    - Templates
        - Designed to formalize macro usage
        - Inspired by Ada generics and Clu's parametrized modules
    - Exception handling
  - Better language for writing and using libraries

Objects

- A packet of information stored in the computer's memory
- Data together with operations on the data (identity, state, and behavior)
- Allow the use of computer as an expressive medium by making the tools look less like machines and more like an extension of our thoughts
- In the physical world, an object can be anything: a car, a person, or an integrated circuit
- Objects have attributes, such as color or size
- Objects exhibit behavior, such as running or changing state in response to a set of inputs

objects = characteristics + behaviors

- Objects are reusable, without having to be redesigned
  - A large portion of the components used on circuit boards for electronic products are standard, inexpensive, off-the-shelf items
  - Availability of such items allows designers to concentrate on solving the problem at hand instead of reinventing the tools with which to solve the problems
- A user can communicate with an object through a set of functions or methods, commonly known as an *interface*
  - Interface is a way of sending a message to the object

Classes

- Objects that are identical (in data characteristics and behavior/functionality) except for their state during a program's execution are grouped together into classes
- Description of the characteristics shared by all objects of the same type
- Abstracted as a data type
  - A floating point number has a set of characteristics and behaviors
  - Provides a *grouping* of object types and operations on those types

- Combination of data and the functions that work on the data
- Defined [by a programmer] to fit a problem
- Allows the extension of a programming language by adding new data types or classes
  - Abstract [user-defined] data types work almost exactly like built-in types
  - User can create variables – objects or instances
  - User can manipulate variables by sending *messages* or *requests*
- Encapsulation
  - Ability to combine both operations and data inside a class
- The standard data types, such as `int`, `double`, and `float`, are known as *intrinsic classes*

Inheritance, or Type relationships

- Expresses similarity between types with the concept of base types and derived types

  Base class. Contains all the characteristics and behaviors that are shared among the types derived from it; created to represent core of the ideas about objects in the system

  Derived class. Created from base class to express the different ways that core can be realized, and is a superset of the base class
- Allows objects to acquire the attributes and behaviors of other objects
- Contributes to economical and maintainable design, because objects share attributes and behaviors without separately duplicating the program code that implements them
- Example of inheritance from biology
  - Taxonomic scheme used by biologists to classify living things
  - Subdivides the plant and animal kingdoms into groups called *phyla*
  - Each phylum is subdivided into *classes*, *orders*, *families*, and so on
  - Lower level groups **inherit** (or share) the characteristics of higher level groups
  - Wolves belong to the canine family implies
    - Wolves have well developed hearing and smell
    - Wolves are carnivores
    - Wolves are a type of mammals, implying that
      - Wolves have hair that regulate their body temperature
      - Wolves are vertebrates and have a backbone
  - Software objects like *windows*
    - Occupy a specific x-y position on computer screen
    - Have a specific height, width, border style, and background color
    - A *menu* object is a window
    - In addition to all the properties of windows, a menu has
      - Line items
      - Possibly a scroll bar
- With such classes, you don't need intermediate models in large problems as required in procedural languages
  - Type hierarchy is the primary model

  - Direct relation between description of the system in the real world and the description of the system in code, hence the simplicity

Polymorphism

- Single name to denote methods/attributes of different object types
- Multiplicity of forms for a single method
- A powerful technique for generalizing a single behavior across many different kind of objects
- Used in conjunction with the term "late binding"

  Early binding (or static binding) occurs when the addresses of all functions are known when the program is compiled and linked

  Late binding (or dynamic binding) does not require the *binding* of function call with the function address until the call is actually made, at the time of program execution
- Allows for higher level of abstraction in software design since the programmer worries only about specifying actions and not how to implement those actions
- Object treated as a member of its base type instead of being treated as a specific type
  - All vectors (`int`, `double`, `complex`) can be added, sized, and operated on by other operations
  - Programmer need not have to worry about how the individual vector types are operated on
  - Such code is also unaffected by the addition of new types, reducing the cost of software maintenance
  - Since the function is going to refer to a generic operation, the compiler may not know at compile time which piece of code will be executed, hence late binding
  - Compiler just checks that the function exists and performs type checking on the parameters and return values
    - If the compiler does not perform type checking, the language is said to be *weakly typed*
  - Compiler inserts a special code, known as *stub*, in place of absolute call to the function; this code contains information to call the right function at run time
- The flexibility of late binding is provided by declaring the function as `virtual`
  - `virtual` functions allow the expression of differences in behavior of classes in the same family
- Manipulating concepts
  - Procedural program in C is a collection of data definitions and function calls
  - Confusing because the terms of expression are more oriented towards computers than the problem being solved
  - In C++
    - Objects represent concepts in the problem space rather than the issue of computer representations
    - Activities are represented as messages sent to those objects
    - Generally less code compared to C because of reuse of existing library code

The first program

- Say hello to the world

```
// Program to say hello to the world

#include <iostream>

int main ( void )
{
    std::cout << "Hello, world" << endl;
    return ( 0 );
}
```

  - You can also use the `'\n'` character for newline just like C
  - The << (put to) operator can take objects such as strings and numbers and send their representation to an *output stream*
    * << is stream has nothing to do with left shift
    * It is like a super-`printf` that can tell the format from the type of operand to be printed
    * Implemented by overloading the << operator
  - `cout` is the standard output stream
  - `//` starts a comment that extends to the end of line
    * The C style comment (`/* ... */`) is also valid and may be useful in certain contexts
  - The `main()` function – As per C standard

    > An implementation shall not predefine the main function. This function shall not be overloaded. It shall have a return type of type `int`, but otherwise its type is implementation-defined. All implementations shall allow both of the following definitions of main:
    > `int main() { /* ... */ }`
    > and
    > `int main ( int argc, char* argv[] ) { /* ... */ }`
  - `endl` is a stream manipulator, and results in the stream being flushed after putting the newline character

- Program is saved as `hello.C`
  - You can save it with the extension `.c`, `.cpp`, or `.cxx`
  - `.C` makes it easier to distinguish it from regular C programs that have the extension `.c` and allow us to use appropriate compiler in the `Makefile` using the suffix rules
  - The compiler to be used on UMSL systems is g++

A cheap calculator

- Write a program that can read in integers and operands from the keyboard, print the answer, and exit
- The input will be given as

  `6 - 5 * 6 / 3 =`

- We'll evaluate each operation on same precedence and from left to right
- The code is:

```
// A cheap calculator

#include <iostream>
```

```
int main ( void )
{
    int x, y;
    char op;

    // Read in the first operand and operator

    cin >> x >> op;

    // Evaluate untill you encounter =

    while ( op != '=' )
    {
        // Read in next operand

        cin >> y;

        switch ( op )
        {
        case '+': x += y; break;
        case '-': x -= y; break;
        case '*': x *= y; break;
        case '/':
            if ( y )
                x /= y;
            else
            {
                cout << "Attempt to divide by zero" << endl;
                exit ( 1 );
            }
            break;
        default:
            cout << "Unknown operand" << endl;
            exit ( 1 );
        }

        // Read in the next operator

        cin >> op;
    }

    // Out of while loop; print result

    cout << "The answer is: " << x << endl;
    return ( 0 );
}
```

- Looks very much like a C program except for the use of `cin` and `cout`
- >> is the *get* operator to read from a stream
  - Automatically discards white space
  - Do not have to use the ampersand like in `scanf` in C
  - Chaining of variables in `cin` and `cout` is a natural consequence of C++ features and no special knowledge needs to be hardwired in the C++ compiler

Methods

- Set of processes and heuristics used to break down the complexity of a program
- The name given in object-oriented languages to a procedure or routine associated with one or more classes
- An object of a certain class knows how to perform actions, e.g. printing itself or creating a new instance of itself, rather than the function (e.g. printing) knowing how to handle different types of object
- Different classes may define methods with the same name (i.e. methods may be polymorphic)
- The term "method" is used both for a named operation, e.g. "PRINT" and also for the code which a specific class provides to perform that operation
- Methods integrate analysis, design, and documentation
- Successful methods
  - Help you to analyze and design (communicate to team members)
  - Don't impose overheads without short-term paybacks (visible progress toward the goal)
- The goal of an OOP method should be to generate a good design
- **Five stages of object design**
  1. Object discovery
     - Occurs during initial analysis of a project
     - Objects must be discovered by looking for external factors and boundaries, duplication of elements in the system, and the smallest conceptual units
     - Commonality between classes suggesting base classes and inheritance may appear right away, or later in the design process
     - *Let a specific problem generate a class, then let the class grow and mature during the solution of other problems*
  2. Object assembly
     - While building objects, you may discover need for new members that didn't appear during discovery
     - Internal needs of an object may require new classes to support it
     - *Discovering the classes you need is the majority of the system design; if you already have the classes, this is a trivial project*
  3. System construction
     - We may have more requirements for an object at this stage
     - Need for communication and interconnection with other objects may change the need for classes or require new classes
     - *Don't force yourself to know everything at the beginning; learn as you go*
  4. System extension
     - Restructuring parts of the system, possibly adding new classes
     - *Start programming; get something working so that you can prove or disprove your design; classes partition the problem and help impose structure*
  5. Object reuse
     - Shortcomings may appear in classes due to reuse
     - Change a class to adapt to new programs, generalizing its definition
     - *Follow the KISS principle; little clean objects with obvious utility are better than big complicated interfaces; Start simple and expand the class interface when you understand it better*

Fractional calculator based on reverse polish notation

- Reverse polish notation
  - Based on the use of a stack to perform calculations
  - Operators act on the numbers most recently pushed on the calculator
  - The sequence of numbers can be entered as

    `1 2 3 * + =`

  - RPN calculator does not require parentheses; to compute

    $$(1 + 2 \times 4)/(1 + 2)$$

    the user enters

    `1 2 4 * + 1 2 + / =`

- We need a stack to implement a solution to the RPN calculator
- We also need a structure to hold the fractions rather than using the `float` to avoid the round off errors
  - `1 3 / 3 * =` should evaluate to 1 rather than 0.999999
  - Declare the structure as

```
struct Fraction
{
    int num;                    // Numerator
    int denom;                  // Denominator

    // Assign default values

    Fraction ( int n = 0, int d = 1 )
    {
        num = n;
        denom = d;
    }
};
```

  - In C++, we do not have to use `typedef` and the above definition is sufficient to declare a fraction as

    `Fraction x;`

  - Notice the initialization sequence to assign default values as $0/1$
  - Using the initialization sequence, we can declare and initialize fractions with

```
Fraction f ( 1, 3 );        // initialized as 1/3
Fraction t ( 2 );           // initialized as 2/1
Fraction r;                 // initialized as 0/1
```

- Operator overloading
- Notice that there is no possibility of "divide by zero" error
  - Divide $1/2$ by $0/1$
  - The result is a structure with the numerator 1 and denominator 0
  - The pairs of integers are not fractions but *models* of fractions
  - Providing reasonable models of objects and actions is an essential aspect of programming

Other things to remember when programming in C++

- Two very important ideas
  1. C++ has many guards in the language and possesses features so that you can build in your own guards; the guards are intended to prevent the program being created from losing its structure even after it has been created and is being maintained.
  2. No matter how much analysis you do, there are some things about a system that won't reveal themselves until design time, and more things that won't reveal themselves until a program is up and running. Because of this, it is critical to move fairly quickly through analysis and design to implement a test of the proposed system. Because of Point 1, this is far safer than when using procedural languages, because the guards in C++ are instrumental in preventing the creation of "spaghetti code."

Comparison with C

Basic data types

- C++ has all the built in types available in C, including `int`, `char`, `float`, and `double`
- Additionally, C++ also has a Boolean type `bool`, and a `void` type that indicates the absence of type information
  - `void` is syntactically a fundamental type but there are no objects of type `void`
  - The following is an incorrect usage of `void`

```
void x;      // It is not possible to have an object of type void
```

  - The following are valid uses of `void`

```
void foo ( void );          // foo is a function that takes no parameters and
                            // returns no value
void * ptr;                 // A void pointer to hold a generic address
```

- The most common floating type operator is `double`, not `float`
  - The math library functions use `double`
- Type of a character constant is `char`; in C, it is an `int`
  - Implicit conversion of a `char` to `int` is allowed, so the following is legal

```
int n += 'A'
```

- The typecasting is done by using a function-like notation such as `char( n )` instead of `( char )n`
  - Exception – The typecasting notation for pointer types is retained and `( char* )p` cannot be written as `char*( p )`
  - The notation `( char* )( p )` is preferable

Variable and constant declarations

- For most of the intent and purpose, the C style of variable and constant declarations is retained
- Every identifier has a type associated with it
  - The type determines the set of operations applicable to the identifier
- The C++ `const` differs slightly from its C counterpart

  - When applied to a basic type, `const` is inline-replaced with the constant and no storage is allocated
  - It is preferable to use `const` rather than `#define` because it is type safe
    * C++ compiler checks the syntax of the `const` statements immediately; the `#define` directive is not checked until the macro is used
    * `const` uses C++ syntax while `#define` uses a syntax of its own
    * `const` uses normal C++ scope rules while constants defined by a `#define` directive continue on forever
  - If the `const` object is global in scope, or its address is taken, the C++ compilers do not substitute its value in place of the name
  - In C++, `const` must be initialized at the same time as they are declared, unless they are declared external

```
const int foo;                  // Invalid, not initialized
extern const int foo;           // Valid, external constant
const int answer = 42;          // Valid, internal and initialized
```

- Enumerated types can be defined with an `enum` declaration

```
enum boolean { false, true };
enum color   { red = 4, green = 2, blue = 1 };
```

  - In C, there is no type checking for enumerations
    * `boolean`, as defined above, is not distinct from `int` but merely a synonym for it
    * A `boolean` variable can hold any integer value, such as -1, 2, or `red`
  - In C++, explicit casts must be used to convert between `enum` and from `int` to `enum`
    * The conversion from `enum` to `int` is still automatic

Arithmetic

- All the arithmetic operators in C are used with the same syntax and semantics in C++

Assignment

- Same syntax and semantics as C
- The use of assignment expressions within other expressions is allowed

Relational and Boolean operations

- Same syntax and semantics as C
- C++ does have a Boolean type/class, known as `bool`, with predefined `true` and `false`
  - Boolean class is used as shown below for variables and functions:

```
bool b = x == y;
bool greater ( const int a, const int b ) { return ( a > b ); }
```

  - By definition, `true` has a value 1 and `false` has a value 0 when converted to integers
  - Conversely, `int`s can be implicitly converted to `bool` with 0 converting to `false` and non-zero values converting to `true`
  - A pointer can be implicitly converted to `bool`, with `null` pointers converting to false and non-`null` pointers converting to `true`

Function declaration

- Same as ANSI C (as we studied)
- If a function does not take any arguments, you must still supply the parentheses, such as

```
x = rand();    // get a random number
```

  - It is legal (but uncommon) to omit the parentheses
  - In such a case, the function name denotes the *starting address* of the function and *not* the return value
  - This address can be passed to other functions as

```
x = integral( -1, 1, exp );   // pass address of exp
```

- If a function does not have arguments, you do not have to put `void` in the header
- The parameters are still passed by value, except arrays which are passed by reference
- If we need to pass parameters by reference, we have to use a *reference* argument, as illustrated by the following function to swap two integers

```
void swap ( int& x, int& y )
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

  - In C, we have to use the `*` and `&` operators to achieve the same effect

- You can still use `const` to avoid modification of parameters passed by reference (like arrays)
- `inline` functions
  - We can instruct the compiler not to generate code for a function call but to replace the function invocation with the actual function
  - Useful to write functions that would have required macros
  - Example

```
inline int max ( const int m, const int n ) const
{
    return ( m > n ? m : n );
}
```

  - Provide a run-time versus code-time trade off
  - Much safer than preprocessor macros
    - The inline function `max ( a[n++], b )` truly computes the maximum of `a[n]` and `b`, and increments `n` once
    - The preprocessor macro

```
#define max( x, y ) ( ( x ) > ( y ) ? ( x ) : ( y ) )
```

      causes computation of the expression `a[n++] > b ? a[n++] : b` which can increment `n` twice
    - The macro is typeless and can compute the maximum of `doubles` as well while inline function cannot do so

- Linking with C functions
  - The prototypes of C functions must be declared with the `extern "C"` qualifier
  - This is done to prevent the *mangling* of function name by the C++ compiler
    - Name mangling is the automatic extension of a function name to distinguish between two functions with the same name (polymorphism)

Control flow

- All control flow statements including `if ... else`, `while`, `do ... while`, `for`, `switch` have the same syntax and semantics as C

Variables

- Variables can be declared anywhere in a function
- Their scope ranges from the point of declaration to the end of the function
- Statements such as

```
for ( int i( 0 ); i < n; i++ ) ...
```

  are valid and preferable to the C style
  - In the above, the scope of `i` is to the end of the `for` statement
- By default, the variables are `automatic` but can be declared as `static`, the behavior being exactly like in C
  - Static variables that are not explicitly initialized are automatically initialized to 0
  - Automatic variables that are not properly initialized contain garbage

Pointers and Arrays

- For most practical purposes, the syntax and semantics for pointers are the same as that in C
- The use of `const` in conjunction with pointers has two meanings

```
char * const q = "String";    // q may not be modified but the string may be
const char * p = "String";    // p may be modified but the string may not be
const char * const r = "string";     // Neither r nor string can be modified
```

- The equivalence between arrays and pointers in C is retained in C++
- The use of `int []` in the function parameters instead of `int *` is encouraged even though both are legal

References

- Basically introduced in C++ to facilitate parameter passing by reference
- Let us consider the function to swap two integers again

```
void swap ( int& x, int& y )
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

- Behind the scenes, all references are translated to pointers, according to the following two rules:
  1. Whenever a reference variable `p` is used, replace `p` by `*p`
  2. Whenever a reference variable is initialized with `a`, replace `a` by `&a`
- You can directly initialize a reference variable as

```
double& p = m;    // same as double *p = &m;
```

- The call to `swap` itself is modified to

```
swap ( &x, &y )
```

- You can also write a function that returns a reference
  - Consider the following function to access an element in a diagonal matrix, assuming that the number of rows in the matrix is the same as the number of columns and every non-diagonal element is a zero

```
double diag[N];    // Diagonal of an N x N matrix, N is odd

double elem ( int i, int j )
{
    return ( i == j ? diag[i] : 0 );
}
```

  - Everything works great unless you try to assign a value to an element

```
elem ( m, n ) = pi;
```

    because `elem ( m, n )` is not an lvalue
  - We can rewrite the function as

```
double& elem ( int i, int j )
{
    return ( i == j ? diag[i] : ... );
}
```

  - Since the function returns a `double&`, it actually returns the address of `diag[i]` and not the value contained therein
  - The problematic assignment above is replaced by

```
*(returned pointer) = pi;
```

  - Problem occurs if we attempt to return a 0 for non-diagonal element because 0 is not an address
  - We can explicitly create a variable for 0 and return a reference to it

```
double zero = 0.0;

double& elem ( int i, int j )
{
    if ( i == j )
        return ( diag[i] );
    else
        if ( zero != 0.0 )
        {
            cerr << "Illegal access occurred previously" << endl;
            zero = 0.0;
        }
    return ( zero );
}
```