cis15
advanced programming techniques, using c++
lecture # II.1

topics:

- objects and class design
- command line arguments

resources:

- Pohl, chapter 4

aggregate data types

- `class` and `struct`
- `struct` comes from C
- `class` is new in C++
- both are aggregate types, meaning that they group together multiple fields of data
- for example:

```
struct point {
  double x, y;
};
```

- don't forget to put a semi-colon at the end of the structure definition!
- in C, the tag (`point`) is optional and does not constitute a data type (you need to use `typedef` as well)
- but in C++, the tag is considered a data type, hence the above example is a data type definition
- which means that you can use `point` as a data type, e.g.:


```
point p;
```

- the fields or elements of an aggregate data type are called *members*
- and they are referred to using "dot notation", e.g.:

```
p.x = 7.0;
p.y = 10.3;
```

- you can also use a *pointer* to access members of an aggregate data type, e.g.:

```
p->x = 12.3;
```

but we will discuss pointers in the next unit, so don't worry about this now...

- you can also declare a structure and variable in the same statement, e.g.:

```
struct {
  double x, y;
} myPoints[3] = { {1, 2}, {3, 4}, {5, 6} };
```


member functions

- in C++, members of aggregate data types can be functions
- (C only allows data members)
- in object-oriented programming (OOP) lingo, the word "method" is often used instead of "function"
- the reason to define functions inside an aggregate data type is to follow the OOP principle of *encapsulation*—operations should be packaged with data
- for example:

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

struct point {
  double x, y;
  void print() const {
    cout << "(" << x << "," << y << ")\n";
  }
```

```
  void set( double u, double v ) {
    x = u;
    y = v;
  }
}; // end of struct--don't forget semi-colon!

int main() {
  point w;
  w.set( 1.2, 3.4 );
  cout << "point = ";
  w.print();
}
```

- notes:
  - `const` keyword in definition of print method indicates that the data members will not be modified inside the method
  - notice that the set method changes the values of the data members—this is considered good OOP practise
  - defining the methods inside the `struct` definition is called "in-line declaration"; this is generally only okay for short, concise methods

- the *class scope* operator can be used when in-line declarations are inappropriate
- for example:

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

struct point {
  double x, y;
  void print() const;
  void set( double u, double v );
}; // end of struct--don't forget semi-colon!

void point::print() const {
  cout << "(" << x << "," << y << ")\n";
} // end of print()

void point::set( double u, double v ) {
  x = u;
  y = v;
} // end of set()
```



```
int main() {
  point w;
  w.set( 1.2, 3.4 );
  cout << "point = ";
  w.print();
} // end of main()
```


public and private access

- members of structures can be `public` or `private`
- `public` means that any code can access the members
- `private` means that only code inside the class or struct can access the members (or "friend" classes, to be discussed later in the term)
- typically, following good OOP practice, all data members are `private` and only function members are `public` (but not all—only those that need to be accessed outside of the struct or class)
- for example:

```
struct point {
public:
  void print() const;
  void set( double u, double v );
private:
  double x, y;
}; // end of struct--don't forget semi-colon!
```

classes vs structs

- in C++, keyword `class` is introduced
- the difference between structs and classes is:
  in a `struct`, the members are `public` by default
  in a `class`, the members are `private` by default
- for example:

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

class point {
  double x, y;
public:
  void print() const;
  void set( double u, double v );
}; // end of class--don't forget semi-colon!
```



```
void point::print() const {
  cout << "(" << x << "," << y << ")\n";
} // end of print()

void point::set( double u, double v ) {
  x = u;
  y = v;
} // end of set()

int main() {
  point w;
  w.set( 1.2, 3.4 );
  cout << "point = ";
  w.print();
} // end of main()
```

- otherwise, class and struct are *syntactically* the same
- but by convention, C++ programmers tend to use `class`

class scope

- the class scope operator is two colons (`::`)
- the `::` operator has the highest precedence in the language, so it always gets evaluated first
- there are two versions of the operator: binary and unary
- we already saw the binary version: `point::print()`, which is used to refer to a variable's "class scope" (also called "local scope")
- the unary version is like this: `::count` and is used to refer to a variable's "external scope" (e.g., for a global variable)
- example from the book:

```
int count = 0; // delcare global variable

void how_many( double w[], double x, int& count ) {
  for ( int i=0; i<N; ++i ) {
    count += ( w[i] == x ); // local count
  }
```



```
  ++::count; // global count
} // end of how_many()
```

- this is only necessary since `count` is declared twice
- if you didn't have the `::count`, then the second time, it would also refer to the local variable
- it is better practise not to use global variables; or at least if you do, give them unique names to avoid confusion :-)

nested classes

- classes can be nested
- example from the book:

```
char c;  // global scope

class X {
  public:
    char c;   // local scope in class X
    class Y {
      public:
        void foo( char e ) { X t; ::c = t.c = c = e; }
      private:
        char c; // local scope in class Y
    };
};
```

- the scope of the third (last) c is `X::Y::c`

"this" pointer

- the keyword `this` is used to refer to an instance of a class from within itself
- it is a *pointer* — something we will discuss at length in the next unit
- for example:

```
point inverse() {
  x = -x;
  y = -y;
  return (*this);
}
```

- this function returns a pointer to itself, i.e., the address of the object in memory
- we'll come back to this when we discuss pointers

"static" members

- the `static` keyword is used to refer to members that do not need to be instantiated
- in other words, it is independent of any class variable
- for example:

```
class point {
  public:
    static int dimensions;
    ...
};
...
int main() {
  ...
  point::dimensions = 2; // initialize but never instantiate point
  ...
}
```


"const" members

- members with the `const` keyword in their definition cannot be modified
- this refers either to data members or to function members to indicate that the data members contained therein are not modified
- for example:

```
class point {
  double x, y;
  public:
    const int dimensions ;
    void print() const;
};
void point::print() const {
  cout << "(" << x << "," << y << ")\n";
} // end of print()
```

- note that the `mutable` keyword can override this
- for example:

```
  mutable int delta;
```

means that even if `delta` is referenced inside a `const` function, its value can be modified. Example:

```
class person {
public:
  void setName( string n ) { name = n; }
  void setAge( int a ) { age = a; }
  void setSSN( unsigned long ssn ) { soc_sec = ssn; }
  void bday() const { ++age; }
  void print() const {
   cout << name << " is " <<  age << " years old. SSN : "
  << soc_sec << endl;
private:
  mutable int age;  // always modifiable
  unsigned long soc_sec;
  string name;
}
```



```
int main(){
  const person ira;
  ira.setName("Ira Pohl");
  ira.setAge(38);
  ira.setSSN(1110111);

  ira.print();
  ira.bday(); // ira.age is mutable
  ira.print();
}
```


special types of classes: "containers"

- there are several special types of classes in C++
- the first we will discuss is called a *container*
- it is a class designed to hold large numbers of objects
- for example:

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

class ch_stack {
  public:
    void reset() { top = EMPTY; }
    void push( char c ) { s[++top] = c; }
    char pop() { return s[top--]; }
    char top_of() const { return s[top]; }
    bool empty() const { return( top==EMPTY ); }
    bool full() const { return( top==FULL ); }
  private:
```

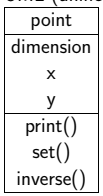


```
    enum{ max_len = 100, EMPTY = -1, FULL = max_len - 1 };
    char s[max_len];
    int top;
};

int main() {
  ch_stack s;
  char str[40] = { "hello world!" };
  int i = 0;
  cout << "str=" << str << endl;
  s.reset();
  while( str[i] && ! s.full() ) {
    s.push( str[i++] );
  }
  cout << "reversed str=";
  while ( ! s.empty() ) {
    cout << s.pop();
  }
  cout << endl;
} // end of main()
```

class design

- data members should be `private` ("hidden")
- function members are often `public` (but not always—private function members can be used for computations internal to a class)
- functions that do not modify data members should be `const`
- pointers add indirection (we'll talk about that later)
- a uniform set of functions should be included: `set()`, `get()`, `print()`
- UML (unified modeling language) provides a graphical method for representing classes



command-line arguments

- example:

```
#include <iostream>
using namespace std;
int main( int argc, char **argv ) {
  cout << "argc = " << argc << endl;
  for ( int i=0; i<argc; i++ ) {
    cout << "[" << i << "]=" << argv[i] << endl;
  }
} // end of main()
```

- executed from the unix command-line like this:

```
unix$ ./a.out asdf 45
argc = 3
[0]=./a.out
[1]=asdf
[2]=45
```