cs3157: c++ lecture #2 (mon-11-apr-2005)

- today:
  - language basics: identifiers, data types, operators, type conversions, branching and looping, program structure
  - data structures: arrays, structures
  - pointers and references
  - I/O: writing to the screen, reading from the keyboard, `iostream` library
  - functions: defining, overloading, inlining, overriding
  - classes: defining, scope, ctors and dtors
  - listing of keywords

chronology of some programming languages...

- (1958) Algol created – the first high-level structured language with a systematic syntax
- (1969) UNIX created using BCPL (Basic Combined Programming Language)
- (1969) B created by Ken Thompson, as a replacement for BCPL
- (1970) Pascal established as the successor to Algol
- (1973) C completed, and released as the successor to B, giving the user control of data types
- (1979) Bjarne Stroustrup begins work on C-with-Classes, an Object Oriented version of C
- (1983) C-with-Classes redesigned and released as C++
- (1985) First mass release of C++ compilers

C++ vs Java

- advantages of C++ over Java:
  - C++ is very powerful
  - C++ is very fast
  - C++ is much more efficient in terms of memory
  - compiled directly for specific machines (instead of bytecode layer, which could also be seen as a portability advantage of Java over C++...)
- disadvantages of C++ over Java:
  - Java protects you from making mistakes that C/C++ don't, as you've learned now from working with C
  - C++ has many concepts and possibilities so it has a steep learning curve
  - extensive use of operator overloading, function overloading and virtual functions can very quickly make C++ programs very complicated
  - shortcuts offered in C++ can often make it completely unreadable, just like in C

identifi ers.

- i.e., valid names for variables, methods, classes, etc
- just like C:
  - names consist of letters, digits and underscores
  - names cannot begin with a digit
  - names cannot be a C++ keyword
- literals are just like in C with a few extras:
  - numbers, e.g.: `5, 5u, 5L, 0x5, true`
  - characters, e.g., `'A'`
  - strings, e.g., `"you"` which is stored in 4 bytes as `'y', 'o', 'u', '\0'`

data types.

- simple native data types: `bool, int, double, char, wchar_t`
- `bool` is like `boolean` in Java
- `wchar_t` is "wide char" for representing data from character sets with more than 255 characters
- modifiers: `short, long, signed, unsigned`, e.g., `short int`
- floating point types: `float, double, long double`
- `enum` and `typedef` just like C

operators.

- same as C, with some additions
- if you recognize it from C, then it's pretty safe to assume it is doing the same thing in C++

type conversions.

- all integer math is done using `int` datatypes, so all types (`bool, char, short, enum`) are promoted to `int` before any arithmetic operations are performed on them
- mixed expressions of integer / floating types promote the lower type to the higher type according to the following hierarchy:

```
int < unsigned < long < unsigned long
  < float < double < long double
```

- you can do explicit conversions like in C using `(int)`, e.g.
- you can also do explicit conversions using C++ operators:
  - `static_cast` – safe and portable; e.g. `c = static_cast<char>(i);`
  - `reinterpret_cast` – system dependent, not good to use
  - `const_cast` – lets you change a `const` into a modifiable variable
  - `dynamic_cast` – used at run-time for casting objects from one class to another (within inheritance hierarchy); this is sort of like Java but can get really messy and is really a more advanced topic...

branching and looping.

- `if`, `if/else` just like C and Java
- `while` and `for` and `do/while` just like C and Java
- `break` and `continue` just like C and Java
- `switch` just like C and Java
- `goto` just like C (but don't use it!!!)

program structure.

- just like in C
- program is a collection of functions and declarations
- language is block-structured
- declarations are made at the beginning of a block; allocated on entry to the block and freed when exiting the block
- parameters are call-by-value unless otherwise specified

arrays.

- similar to C
- dynamic memory allocation handled using `new` and `delete` instead of `malloc` (and family) and `free`
- examples:

```
int a[5];
char b[3] = { 'a', 'b', 'c' };
double c[4][5];
int *p = new int(5);   // space allocated and *p set to 5
int **q = new int[10]; // space allocated and q = &q[0]
int *r = new int;      // space allocated but not initialized
```


structures.

- `struct` keyword like in C (but you don't need `typedef`)
- use dot operator or `->` to access members (fields) of a `struct` or `struct *`
- C++ allows functions to be members, whereas C only allows data members (i.e., fields)
- example

```
struct point {
  public:
    void print() const { cout << "(" << x "," << y << ")"; }
    void set( double u, double v ) { x=u; y=v; }
  private:
    double x, y;
}
```


pointers and references.

- *pointers* are like C:
  - `int *p` means "pointer to int"
  - `p = &i` means p gets the address of object i
- *references* are not like C!! they are basically aliases – alternative names – for the values stored at the indicated memory locations, e.g.:

```
int     n;
int     &nn = n;
double  a[10];
double  &last = a[9];
```

- the difference between them:

```
int a = 5;       // declare and define a
int *p = &a;     // p points to a
int &ref_a = a;  // alias (reference) for a
*p = 7;          // *p points to a, so a is assigned 7
a = *p + 1;      // a is assigned value of *p=7 plus 1
```

I/O: writing to the screen.

```
// hello world in C++
#include <iostream>
using namespace std;

int main() {
  cout << "hello world" << endl;
}
```

- comment characters are `//` or `/* ... */`, just like Java
- `using namespace` is sort of like importing a package in Java; it is used in conjunction with the header declaration
- you could also say `#include <iostream.h>` and leave out the `using namespace std;` line; this is an older style of C++ but it still works
- `cout <<` is like `System.out.print` in Java or like `printf()` in C
- `endl` outputs a newline; saying `cout << "\n";` does the same thing

I/O: reading from the keyboard.

- read from the keyboard using `cin >>`, which is like `scanf()` in C
- example:

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

int main() {
  int i;
  cout << "enter a number: ";
  cin >> i;
  cout << "you entered " << i <<"\n";
}
```


I/O: C++ iostream.

- two bit-shift operators:
  - `<<` meaning "put to" output stream ("left shift")
  - `>>` meaning "get from" input stream ("right shift")
- three standard streams:
  - `cout` is standard out
  - `cin` is standard in
  - `cerr` is standard error
- if you specify the *namespace*, you can use these directly; otherwise you'd have to say `std::cout`, e.g.
- the *iostream* library is "type safe", so you don't have to use formatting statements — variables are input/output based on their datatype

I/O: ostream and istream.

- `ostream`
  - `cout` is an `ostream`, `<<` is an operator
  - use `cout.put( char c )` to write a single char
  - use `cout.write( const char *p, int n )` to write n chars
  - use `cout.flush()` to flush the stream
- `istream`
  - `cin` is an `istream`, `>>` is an operator
  - use `cin.get( char &c )` to read a single char
  - use `cin.get( char *s, int n, char c='\n' )` to read a line (inputs into string `s` at most `n-1` characters, up to the specified delimiter `c` or an `EOF`; a terminating `0` is placed at the end of the input string `s`)
  - also `cin.getline( char *s, int n, char c='\n' )`
  - use `cin.read( char *s, int n )` to read a string

I/O: formatted output.

- in <iomanip> header file, the following are defined:
- `scientific` – prints using scientific notation
- `left` – fills characters to right of value
- `right` – fills characers to left of value
- `internal` – fills characters between sign and value
- `setfill( int )` – sets fill character
- `setw( int )` – sets field width
- `setprecision( int )` – sets floating point precision

I/O: fstream.

- definitions:

```
ifstream();
ifstream( const char *,int ios::in, int prot=filebuf::openprot );
ofstream();
ofstream( const char *,int ios::out, int prot=filebuf::openprot );
```

where

```
ios::in = open for input
ios::app = open for appending
ios::out = open for output
```

- if you create an `ifstream` or `ofstream` using a default constructor, then use `open( const char *,int ios::in, int prot=filebuf::openprot );` to open the associated file
- use `close()` to close the file when done with it
- use `put()` to send output to `ofstream` and use `get()` to get input from `ifstream`

functions.

- parameters are call-by-value
- function prototypes are just like in C
- default arguments
  - used when a function is frequently called with the same argument value
  - defined in the function header (and prototype)
  - e.g., `int add_increment( int i, int increment = 1 );`
    invoked as either:
    `i = add_increment( j );` to add 1 to j or
    `i = add_increment( j, x );` to add x to j
- functions can be arguments (like function pointers in C) (advanced topic...)

functions: overloading

- like in Java
- when you use the same name for functions with different signatures
- i.e., allows multiple definitions for the same function name within the same scope, with different variable types
- e.g.:

```
double average( const int size, int& sum );
double average( const int size, double& sum );
```

functions: inlining

- same principle as `#define` macros
- but type safe
- purpose is to save runtime by avoiding function invocation if/when possible
- example:

```
#define CUBE(X) ((X) * (X) * (X))
```

versus

```
inline double cube ( double x ) {
  return( x * x * x );
}
```


functions: chaining

- calling the base-class version of a function from the derived class
- suppose both `IntArray` and `StatsIntArray` have functions called `init()`
- you can invoke `IntArray::init()` from `StatsIntArray::init()`:

```
void StatsIntArray::init() {
  IntArray::init(); // chaining
  buf = 0;
}
```


classes.

- `class` keyword
- just like `struct` except with `class`, the default privacy specification is `private` whereas with `struct`, the default privacy specification is `public`
- example

```
class point {
  double x, y; // implicitly private
  public:
    void print() const { cout << "(" << x "," << y << ")"; }
    void set( double u, double v ) { x=u; y=v; }
}
```

- classes can be nested
- `this` is like in Java
- `static` is like in Java, with some wierd subtleties

classes: function overloading and overriding

- overloading:
  - when you use the same name for functions with different signatures
  - functions in derived class supercede *any* functions in base class with the same name
- overriding:
  - when you change the behavior of base-class function in a derived class
  - DON'T OVERRIDE BASE-CLASS FUNCTIONS!!
    - because compiler can invoke wrong version by mistake
    - but `init()` is okay to override
    - (more explanation in ch 12...)

classes: class scope operator.

- `::`
- example:

```
::i        // refers to external scope
point::x   // refers to class scope
std::count // refers to namespace scope
```

- given previous definition of `point`, we could do:

```
point p;
p.print();
p.point::print(); // redundant but legal
```


classes: constructors and destructors.

- constructors are called *ctors* in C++; they take the same name as the class in which they are defined, like in Java
- destructors are called *dtors* in C++; they take the same name as the class in which they are defined, preceded by a tilde (˜); sort of like finalize in Java
- ctors can be overloaded and can take arguments
- dtors can not
- default constructor has no arguments
- constructor with one argument is a *conversion constructor* that converts its argument datatype to an object of the class being constructed
- constructor initializer is a special type of constructor that is used to initialize the values of data members of a class

- example:

```
class point {
  public:
    point() : x(0), y(0) { } // default
    point( double u ) : x(u), y(0) { } // conversion
    point( double u, double v ) : x(u), y(v) { }
.
.
.
}
```


classes: more about constructors

- default constructor (ctor")
- has same name as class it constructs
- in `array5.cpp`, ctor is used instead of `init()`
- declare as:

```
class IntArray() {
public:
  IntArray();
//  etc
}
void IntArray::IntArray() {
  numElems = 0;
  elems = 0;
} // end of default constructor
```

- invoked when object is allocated: `IntArray a;`
- but remember that built-in types are not automatically initialized

more about destructors

- default destructor ("dtor")
- performs same job as `cleanup()`:

```
class IntArray {
public:
  IntArray(); // constructor
  ~IntArray(); // destructor
  // etc
}
void IntArray::~IntArray() {
  if ( elems != 0 ) free( elems );
}
```

- invoked automatically when object is no longer usable (i.e., when it is popped off the stack, like a local function variable)

more stuff to know about ctors and dtors

- chaining
  - constructors and destructors are chained automatically
  - derived class ctors invoke base class constructors and
  - execute in reverse order (lowest base class first)
  - derived class dtors invoke base class dtors and execute in order (derived class first)
- arrays
  - default ctors and dtors are called on each element in the array
- implicit ctors and dtors exist (and are invoked) if you don't write them explicitly
- ctors and dtors can be `private`, but typically are `public`
- never invoke default ctors or dtors explicitly!
  e.g.: `ia.IntArray();  // NO!!!`
  `ia.~IntArray(); // NO!!!`
- stages of object's life (p 75)

classes: abstraction with member functions

- example #1: `array1.cpp`
- example #2: `array2.cpp`
  - `array1.cpp` with interface functions
- example #3: `array3.cpp`
  - `array2.cpp` with member functions
- `class` definition
- `public` vs `private`
- declaring member functions inside/outside class definition
- scope operator (`::`)
- `this` pointer

classes: access specifi ers

- `public`
  - `public` members
  - can be accessed from any function
- `private`
  - `private` members
    - can only be accessed by class's own members
    - and by "friends" (see ahead)
  - "access violations" when you don't obey the rules...
- can be listed in any order
- can be repeated

classes: friends

- allows two or more classes to share private members
- e.g., container and iterator classes
- friendship is not transitive

classes: hierarchy with composition and derivation

- composition:
  - creating objects with other objects as members
  - example: `array4.cpp`
- derivation:
  - defining classes by expanding other classes
  - like "extends" in java
  - example:

```
class SortIntArray : public IntArray {
    public:
      void sort();
    private:
        int *sortBuf;
    }; // end of class SortIntArray
```

  - "base class" (`IntArray`) and "derived class" (`SortIntArray`)
  - derived class can only access public members of base class

  - complete example: `array5.cpp`
  - `public` vs `private` derivation:
    - `public` derivation means that users of the derived class can access the public portions of the base class
    - `private` derivation means that all of the base class is inaccessible to anything outside the derived class
    - `private` is the default

classes: derivation, continued.

- encapsulation
  - derivation maintains encapsulation
  - i.e., it is better to expand `IntArray` and add `sort()` than to modify your own version of `IntArray`
- friendship
  - not the same as derivation!!
  - example:
    - $b2$ is a friend of $b1$
    - $d1$ is derived from $b1$
    - $d2$ is derived from $b2$
    - $b2$ has special access to private members of $b1$, as a friend
    - but $d2$ does not inherit this special access
    - nor does $b2$ get special access to $d1$ (derived from friend $b1$)

classes: derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can't go the other way
- example:

```
  main() {
    IntArray      ia, *pia;
// base-class object and pointer
    StatsIntArray sia, *psia;
// derived-class object and pointer
    pia = &sia;  // okay: base pointer -> derived object
    psia = pia;  // no: derived pointer = base pointer
    psia = (StatsIntArray *)pia; // sort of okay now since:
 // 1. there's a cast
 // 2. pia is really pointing to sia,
 // but if it were pointing to ia, then
 // this wouldn't work (as below)
    psia = (StatsIntArray *)&ia; // no: because ia isn't a StatsInt
```



```
}
```

- danger:
  - don't point a base class pointer to an array of derived objects!
  - they aren't the same size!

C++ keywords.

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	