COMMAND LINE ARGUMENTS

Today

- We will recap some C++ basics
  - Type casting
  - Enumeration types
  - `typedef`
  - Precedence and associativity
  - Control flow
- We'll also introduce what is probably a new topic for most of you:
  - Command line arguments

Type casting

- Used to convert between fundamental (simple) data types (e.g., `int`, `double`, `char`)
- There are two ways to do this
- The C way (technically obsolete):

```
double d = 65.0;
int i = (double)d;
char c = (char)i;
```


- The C++ way:
  - `static_cast`: for conversions that are "well-defined, portable, intertable"; e.g., like the C ways, above.
  - `reinterpret_cast`: for conversions that are system-dependent (not recommended).
  - `const_cast`: to create a modifiable copy of a `const` variable; data type into which the value is cast must always be a pointer or reference (see on).
  - `dynamic_cast`: for converting between classes (to be discussed later in the term)

- Syntax:

  `static_cast<`*type*`>(`*variable*`)`

- In practice this looks something like:

```
double d = 65.5;
int    i;
i = static_cast<int>(d);
```

  converts a double to an integer.

- Const casting:

```
const int c = 5;
my_func(const_cast<int&>(c));
```

  passes a modifiable copy of `c` to the function.

- See `cast.cpp`

Enumeration types

- Used to declare names for a set of related items
- For example:
  `enum suit { diamonds, clubs, hearts, spades };`
- Internally, each name is assigned an `int` value.
- The value assigned to the first name is zero.
- The value of each member of the list is then one more than its lefthand neighbor.
- So in the above example, `diamonds` is actually 0, `clubs` is 1, and so on.

- You create an `enum` data type if you want to use the names instead of the values, so you shouldn't really care what the values are internally.
- If you need to set the value explicitly, you can:

  `enum answer { yes, no, maybe = -1 );`

- If you do this you have to be careful about duplicated values (see `enum.cpp`).
- syntax:

  `enum` *tag* `{` *value0, value1, ... valueN* `};`

- The tag is optional.
- You can also declare variables of the enumerated type by adding the variable name after the closing }
- See `enum.cpp`


```
void showSuit( int card ) {

enum suits { diamonds, clubs, hearts, spades } suit;

suit = static_cast<suits>( card / 13 );

switch( suit ) {
  case diamonds: cout << "diamonds"; break;
  case clubs:    cout << "clubs";    break;
  case hearts:   cout << "hearts";   break;
  case spades:   cout << "spades";   break;
 }

 cout << endl;
}
```

typedef

- The `typedef` keyword can be used to create names for data types
- A `typedef` name is just a synonym.
- For example:

```
typedef int numbers;  // "numbers" is my name
typedef char letters; // "letters" is my name
typedef enum suits  { diamonds, clubs,
                      hearts, spades };
```

- Then you use the name you've created (`numbers`, `letters` or `suits` from the example above)

Precedence and associativity

- "Precedence" means the order in which multiple operators are evaluated
- "Associativity" means which value an operator *associates* with, which is particularly good to know if you have multiple operators adjacent to a single variable
- Associativity is either:
  - left to right, e.g., `3 - 2` (subtract 2 from 3)
  - right to left, e.g., `-3` (meaning negative 3)
- Note that $++$ and $--$ can be either:
  - *postfix* operators are left to right (meaning that you evaluate the expression on the left first and then apply the operator)
  - *prefix* operators are right to left (meaning that you apply the operator first and then evaluate the expression on the right)

Precedence and associativity table

(listed in order of precedence)

operator	*associativity*
:: (global scope), :: (class scope)	left to right
[], $->$, $++$ (postfix), $--$ (postfix), `dynamic_cast<type>` (etc)	left to right
$++$ (prefix); $--$ (postfix), !, `sizeof()`, $+$ (unary), $-$ (unary), $*$ (indirection)	right to left
$*$, $/$, $\%$	left to right
$+$, $-$	left to right
$<<$, $>>$	left to right
$<$, $<=$, $>$ $>=$	left to right
$==$, $!=$	left to right
$\&$	left to right
$\wedge$	left to right
$\vert$	left to right
$\&\&$	left to right
$\vert\vert$	left to right
? :	left to right
$=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $>>=$, $<<=$, $\&=$, $\wedge=$, $\vert=$	left to right

See `prec.cpp`

Control flow

- Branching:
  - `if`,
  - `if-else`,
  - `switch`
- Looping:
  - `for`,
  - `while`,
  - `do...while`
- See `control.cpp`

Command-line arguments

- The UNIX commands we looked at last time are just C/C++ programs
- They have a different form of interaction from the programs you wrote for CIS 1.5.
- Command line arguments.

  `g++ myprog.cpp -o myprog.o`
- Turns out that C/C++ makes it easy to write programs like this.

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()
```

- `cmdline.cpp`

- Executed from the unix command-line like this:

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

- So we have a way of passing an arbitrary number of arguments to a program.

- `argc` tells us how many arguments there are.
- (Well, it actually says how many things are typed into the shell program).
- `argv` gives us the arguments.
- `argv` is (roughly speaking) an array of strings
  - Each thing typed into the shell is a stored as a string.
- To use the arguments, we have to do some manipulation.
- For example, we use `atoi` to retrieve numerical arguments.

- How would we write a simple calculator?

```
unix> calc + 2 3
unix> 5
unix> calc * 2 4
unix> 8
unix>
```

- It should be able to add, subtract, multiply and divide two integers

Summary

- This lecture finished up our quick revision of the material from CIS 1.5
- We looked at:
  - Type casting
  - Enumeration types
  - `typedef`
  - Precedence and associativity
  - Control flow
  - Command line arguments
- The new thing we covered was the Unix/C++ mechanism for handling command line arguments.