COMP322 - Introduction to C++

Lecture 02 - Basics of C++

School of Computer Science

15 January 2013

C++ basics - Arithmetic operators

Where possible, C++ will automatically convert among the basic types.

```
+       // Addition
-       // Subtraction
*       // Multiplication
/       // Division
%       // Integer remainder
```

Another important operator is the assignment operator:

```
=       // Assignment
```

C++ basics - Comparison operators

The result of a comparison operator is always a value of type 'bool':

```
==      // equal
!=      // not equal
>       // greater than
<       // less than
>=      // greater than or equal
<=      // less than or equal
```

C++ basics - Logical operators

The logical `&&` and `||` operators use short-circuit evaluation. They execute the right hand argument only if necessary to determine the overall value.

```
&&    // logical and
||    // logical or
!     // logical negation
```

C++ basics - Bitwise operators

These operators support logical operations on bits. For example,

```
int x = 0x1001 ^ 0x2001;
std::cout << std::hex << x << std::endl;
```

would print 3000.

```
&     // bitwise and
|     // bitwise or
^     // bitwise exclusive or
~     // bitwise complement
<<    // left shift
>>    // right shift
```

C++ basics - `if` statement

```
// Simplest form
if (response == 'y') return true;

// Less simple
if (result > 0.0) {
  x = 1.0 / result;
  y += x;
}
else {
  std::cout << "Division by zero!";
}
```

C++ basics - switch statement

```
int response;

std::cin >> response; // Get input

switch (response) {
case 'y':
   return true;
case 'n':
   return false;
case 'q':
   exit(0);
default:
   std::cout << "I didn't get that, sorry\n";
   break;
}
```

In C++the switch statement will keep going until it hits a break statement, meaning more than one case can be executed.

C++ basics - `while` statement

```
float array[10];
int i;

i = 0;
while (i < 10) {
    array[i] = 0;
    i++;
}
```

Notice one difference between this array declaration and Java: No use of the new operator. When you create an array in C or C++you are essentially just creating continuous variables. (Unlike in Java, there is no notion that this is now a "reference" type)

Like in C, you should not assume an array values are initialized to anything!

C++ basics - `for` statement

Typically a shorthand for common forms of the `while` statement.

```
float array[10];

for (int i = 0; i < 10; i++) {
    array[i] = 0;
}
```

C++ basics - do while statement

```
int response;
do {
  std::cin >> response;
  processCommand(response)
} while (response != 'q');
```

C++ basics - Identifier scope

```
int v = 1;     // Global scope

int main()
{
  int c = 5;  // Local scope

  // Declare 'i' in statement scope
  for (int i = 0; i < c; i++) {
     // do something
  }
  // 'i' is now undefined
  c = c + v;
}
```

C++ basics - Functions

```
/* Calculate the mean of an array */
double mean(double data[], int n)
{
  double sum = 0.0;  // Initialization
  if (n != 0) return 0.0;
  for (int i = 0; i < n; i++)
    sum += data[i];
  return sum / n;
}

/* Impractical recursive factorial */
long factorial(long t)
{
  if (t <= 1) return 1;
  return t * factorial(t - 1);
}
```

C++ basics - Function orders

In C++, you may not CALL a function before it is defined, unless you have put a prior function header. For example:

```
double bar(int x); //without this, it would not compile!

double foo() {
     bar(3);
}

double bar(int x) {
     .....
}
```

Preprocessor

The C++ preprocessor is inherited from C. It runs before the compiler, processing its directives and outputting a modified version of the input.

```
#define   #include
#ifdef    #ifndef
#if       #elif
#else     #endif
#line     #undef
#error    #pragma
```

Preprocessor - #define

The #define directive is used to define textual substitution or macro. It is often used to create constants:

```
#define PI 3.14159
#define HASHTABLESIZE 100
#define MYNAME "fred"
```

By convention, capital letters are often used.

The construct can be used for more elaborate statements that may accept arguments:

```
#define MAX(a,b) ((a < b) ? (b) : (a))
#define SWAP(a,b,type) {type tmp = a; \
                        a = b; \
                        b = tmp; }
```

Preprocessor - `#include`

The `#include` directive is used to incorporate another file into the current file:

```
#include <iostream>
#include <sys/stat.h>
#include "myhdr.h"
```

By convention, header filenames often end in ".h". However, this is purely optional.

If the filename is enclosed in quotes, the preprocessor searches the local directory, *then* a list of standard directories.

If the filename is enclosed in <>, the preprocessor searches *only* the standard directories.

Preprocessor - #if/#ifdef

The #if, #ifdef and #ifndef directives support conditional compilation:

```
#ifndef PI
#define PI 3.14159
#endif

#if (VERSION >= 2)
// code for version 2 and later
#elif (VERSION >= 1)
// code for version 1
#else
// default code
#endif
```

Preprocessor - #if/#ifdef

The `defined` operator can replace #ifdef and #ifndef:

```
#if !defined PI && defined USEMATH
#define PI 3.14159
#endif
```

These constructs can nest:

```
#if !defined _MYHDR_H
#define _MYHDR_H 1
#if __OSTYPE==LINUX
#define USETHREADS 1
#else
#define USETHREADS 0
#endif // __OSTYPE
#endif // _MYHDR_H not defined
```

Preprocessor - everything else

- `#undef` deletes a macro.

```
#undef PI
```

- `#line` overrides the line number and filename. It is primarily used internally by the compiler.

```
#line 1020 "myfile.cpp"
```

- `#error` aborts the compilation and prints a message:

```
#ifndef BLEH
#error You must define BLEH
#endif
```

- `#pragma` is a used to implement compiler-specific commands.

A few interesting operators

- `sizeof` *expr* - returns the size, in bytes, of the expression or named type.
- $expr_1$, $expr_2$ - the comma operator evaluates both expressions sides, returning the value of $expr_2$.
- $expr_1 ? expr_2 : expr_3$ - If $expr_1$ is non-zero (true), the value $expr_2$. Otherwise, the value is $expr_3$.

```
x = (y > 0) ? (100 / y) : 0;
```

- Assignment - assignments return the assigned value, so we can have odd expressions like:

```
x = (y = 2) * 4;
```

Symbol declarations and definitions

We already discussed the builtin types. Certain keywords can modify the treatments of objects or functions we declare:

- `auto` - internal reference in temporary storage
- `volatile` - the value may changed unexpectedly
- `register` - the value is used often
- `const` - a constant value
- `extern` - external reference
- `static` - internal reference in permanent storage

Of these, only `extern` and `static` apply to function symbols.

auto, volatile, and register

- The auto keyword specifies automatic temporary storage, and is implicit. Within a function,

  ```
  auto float n;
  ```

  is equivalent to:

  ```
  float n;
  ```

- The volatile keyword warns the compiler that the value may change unexpectedly, as in a memory-mapped device:

  ```
  volatile int semaphore;
  ```

- The register keyword informs the compiler that you will use the value often. Compilers may ignore this.

  ```
  register int i; // Loop counter, e.g.
  ```

The const keyword

Any data object can be specified as const. This means you cannot modify it after initialization.

```
const float pi = 3.14159; // Global constant

double circle_area(const double radius) {
  pi = 0;      // Illegal
  radius = 0; // Likewise
  return (pi * radius * radius);
}
```

A const variable *must* be initialized.

```
const float pi; // Illegal
```

The `extern` keyword

The `extern` keyword *declares* an function or object which is *defined* later in the file, or in another file.

```
// Use constant 'c' and make it
//  globally visible:
extern const double c; // No initializer!

double energy(double mass)
{
  // Use pow() locally
  extern double pow(double, double);

  return mass * pow(c, 2);
}
// c is still visible, but pow() is not.
```

The `static` keyword

Outside a function, `static` makes a local variable:

```
static int _hidden = 12;
int _visible = 0;
```

`_hidden` is local, whereas `_visible` is global.

In a function, `static` reserves permanent space for a variable:

```
static void example()
{
  static int initialized = 0;
  if (!initialized) {
    initialized = 1;
    // Perform initialization
  }
  // Perform normal function
}
```

Defining and declaring arrays

Arrays are defined by specifying constant array bounds after the variable name.

Arrays which are either global or static are implicitly initialized to zero. Automatic arrays are *not* implicitly initialized.

```
#define N_ITEMS 10
static float vector[N_ITEMS];
const int n_rows = 10;
const int n_cols = 12;
double array[n_rows][n_cols];
```

If an array is initialized explicitly, or is external, the bounds need not be specified:

```
const char message[] = "Hello, World!";
int table[] = {1,4,9,16,25};
extern float vector[]; // Unknown size!
```

Using arrays

Array indices in C++ always begin at zero. The last index is therefore one less than the declared size of that dimension.

```
const int n_items = 10;
float psqr[n_items];
int i;

for (i = 0; i < n_items; i++) {
  psqr[i] = (i + 1)*(i + 1);
}
```

This loop accesses elements 0 through 9, which is correct.

Function definitions

```
/* Calculate the mean of an array */
double mean(const double data[],
            int n)          // Length of 'data'
{
  double sum = 0.0;  // Initialization
  if (n != 0) return 0.0;
  for (int i = 0; i < n; i++)
    sum += data[i];
  return (sum / n);
}

/* Impractical recursive factorial */
long factorial(long t)
{
  if (t <= 1) return 1;
  return t * factorial(t - 1);
}
```

More about functions

- Function definitions do *not* nest.
- A function may return a value or `void`.
- A function definition specifies the parameters passed by the caller.
- A function may contain local variable definitions.
- The variable declarations are followed by a series of statements.
- Once a function is declared, it can be called:

```
y = sin(PI * x);
printf("%f", y);
```

Inline functions

The overhead of calling a function can be significant.

Using the `inline` keyword requests that the compiler expand small functions, as by macro substitution, rather than through an explicit procedure call:

```
inline double circlearea(double radius)
{
  return PI * radius * radius;
}
```

Like the `register` keyword, the compiler is free to act on this or not.

Default parameter passing

For simple types (*not* arrays), parameters are passed by value.

```
#include <iostream>

void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int x = 2, y = 3;

  swap(x,y);
  std::cout << "x=" << x << ", y=" << y << std:endl;
}
```

What does this program print?

User-defined types

C++ inherits three simple ways to create complex types.

- An `enum` defines a type with named symbolic values.
- A `struct` defines a type which consists of one or more fields of possibly distinct subtypes. All fields coexist simultaneously in the type.
- A `union` defines a type which consists of one or more aliases for the same storage.

Enumerated types

An enum associates a name with an integral value.

```
enum color { red, blue, yellow, green};
enum color dot = red;
```

Implicit enum values begin at zero and increment for each item on the list. So in the above example, "green" would have integral value 3.

This behavior can be overridden:

```
enum lettergrades { A=4, B=3, C=2, D=1 };
```

If an explicit value is not given, implicit numbering begins at the last value given:

```
enum color { red=1, blue, yellow, green};
```

Now "green" would have the value 4.

The struct

A C++ struct contains a list of objects, like a database record.

```
struct symbol {
    enum symtype s_type;
    char s_name[32];
    int s_value;
};

struct symbol sym;
struct symbol symtable[26];
```

We can then access the elements using the '.' operator:

```
sym.s_value = 10;
strcpy(sym.s_name, "x");
```

The union

A union definition looks like a struct definition, but the items all share the same storage:

```
union symvalue {
  float sv_flt;
  int sv_int;
  char sv_str[32];
};
```

We can store *either* sv_flt, sv_int, or sv_str, but only one at a time:

```
float x;
int n;
union symvalue val;

val.sv_int = 10;
n = val.sv_int;  // This is OK!
x = val.sv_flt;  // This may be bad!
```

Typedef statements

A type can be associated with a given symbolic name using the typedef statement:

```
typedef unsigned short word;

// word is now a synonym for 'unsigned short'

word array[100];

typedef float rgb[3];

/* rgb is now a synonym for an array with three
 * floating point elements
 */
rgb pixel;

pixel[0] = 1.0;
pixel[1] = pixel[2] = 0.5;
```

Using namespaces

A namespace supports grouping of a set of functions or objects. For example, most of the standard library is associated with the namespace "std":

```
#include <iostream>

int main() {
  std::cout << "Hello!" << std::endl;
}
```

The scope resolution operator '::' tells C++ we want to use the "cout" associated with the namespace "std". We can simplify our code with the `using` statement:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello!" << endl;
}
```

Using namespaces

We can be more specific about using only those things we actually need:

```
#include <iostream>
using std::cout;
using std::endl;
int main()
{
  cout << "Hello, world!" << std::endl;
  return 0; // Return code for success
}
```

Even if we explicitly import a symbol from a namespace, we can still use the fully-qualified form of the name.

Defining namespaces

```
namespace A {
  int factorial(int x) {
    return (x <= 1) ? 1 : (x * factorial(x - 1));
  }
}

namespace B {
  int factorial(int x) {
     int y = 1;
     for (int i = 1; i < x; i++)
       y *= x;
     return y;
  }
}

int main()
{
  cout << A::factorial(5) << endl;
  cout << B::factorial(5) << endl;

  using namespace A;

  cout << factorial(5) << endl;
}
```