COMP322 - Introduction to C++

Winter 2011

Lecture 2 - Language Basics

Milena Scaccia

School of Computer Science
McGill University

January 11, 2011

Course Web Tools

- Announcements, Lecture Notes, Assignments will be posted at:
  `http://www.cs.mcgill.ca/~mscacc/comp322/`
- Assignment Submissions, Discussions
  `http://www.mcgill.ca/mycourses`

Key Dates

- **08 Feb** - Assignment 1 Due
- **15 Feb** - In-Class Test 1
- **22 Feb** - Study Break - No Class
- **29 Mar** - In-Class Test 2
- **05 Apr** - Assignment 2 Due

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;
  return 0;            //Return code for success
}
```

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;
  return 0;            //Return code for success
}
```

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.

User-defined 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;
}
```

Side note: Difference with C

In C, we use the function `fprintf` (or one of its relatives) to write formatted data:

```
#include <stdio.h>

printf("Hello World!\n");
fprintf(stdout, "Hello World!\n");

int n = 42;
fprintf(stdout, "%d\n", n);
```

C++ provides iostreams (not found in the C language):

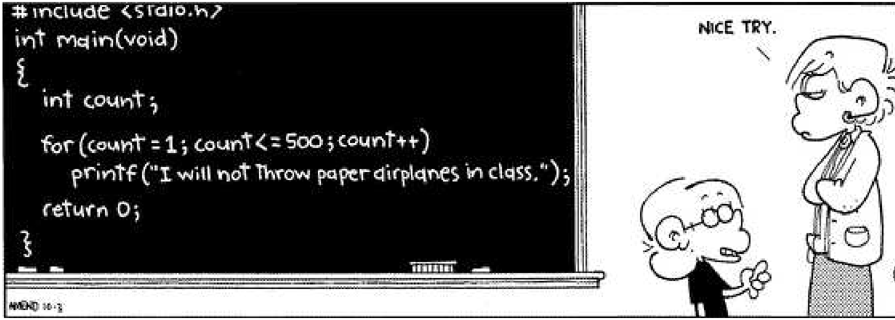
```
#include <iostream>

cout << "Hello World!" << endl;

int n = 42;
cout << n << endl;
```

C++ is very compatible with C, so we can actually use either iostreams or C functions to write data.

C/C++ Code



```c
#include <stdio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

Figure: FoxTrot 10/03/2003

cout vs printf

So what is the difference between cout and printf?

- printf is a C function
- cout is a stream object with '<<' operator to write data
- printf has no type checking
- iostreams are type safe and extensible for user-defined types
- printf returns an integer which denotes the number of characters printed, while cout does not.

Similarly, there is a cin object with operator '>>' for reading from keyboard in C++:

```
int age;
cin >> age;
```

More on the Preprocessor

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

So far, we have seen one example of a preprocessor directive: `#include`.

There are several others:

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

Preprocessor - #include

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

```
#include <iostream>
#include <stdio.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 - #define

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

```
#define PI 3.14159
#define ULTIMATEANSWER 42
#define NAME "Mr. Nobody"
```

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))
```

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
```

The defined operator can replace #ifdef and #ifndef:

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

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 BBQ
  #error You must define BBQ
  #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. The compiler may ignore the `register` keyword.

Example: 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* a function or object which is *defined* later in the file, or in another file. It is used to declare variables of global scope in C++ projects. When the keyword extern is used, the compiler will not allocate memory for the variable.

```
// 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.
```

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 values[n_items];
int i;

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

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

More about functions: 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>
using namespace std;

void half(int a) {
  a = a/2;
}

int main() {
  int x = 200;

  half(x);
  cout << "x=" << x << endl;
}
```

What does this program print?
How can we fix the above so that the value of x is actually modified?

Sneak peak at references

We can use references in order to pass-by-reference instead of by value. This means that if the function alters the data, the original is altered.

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

void half(int &a) { // & is known as a reference operator
  a = a/2;
}

int main() {
  int x = 200;

  half(x);
  cout << "x=" << x <<endl;
}
```

We will revisit references in detail in Lecture 3.

Function Prototypes

A C++ compiler processes the source code from top to bottom. If a function has not already been seen, a compiler would not include its signature in the list of candidates.

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

void half(int &a); // function prototype

int main() {
  int x = 200;

  half(x);
  cout << "x=" << x <<endl;
}

void half(int &a) {
  a = a/2;
}
```

Header File Example

example.h

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

void half(int &a);
```

example.cpp

```
#include "example.h"

int main() {
  int x = 200;

  half(x);
  cout << "x=" << x <<endl;
}

void half(int &a) {
  a = a/2;
}
```

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 patient {
   enum bloodtype ptype;
   char name[32];
   int age;
};

struct patient p1;
struct patient ptable[30];
```

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

```
p1.age = 99;
strcpy(p1.name, "Joe");
```

The union

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

```
union studentdata {
  int idnumber;
  char username[5];
};
```

We can store *either* idnumber, or username, but only one at a time.

A practical use of union is when you would like to allow a single type of data to be accessed in several ways. For example, using the above, we can define and access student data using either their student number (int) or by their username (five chars).

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;

// Define a type point to be a struct with integer members x, y
typedef struct {
  int x;
  int y;
} point;

point p = {1,2};
```

C++ String Library - Examples

See `http://www.cplusplus/com/reference/string/` for more detail.

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  string str;
  string str2 ("cplusplus"); // Declaration

  // Read a string from keyboard
  cout << "Please enter full name: ";
  // getline allows you to read strings with whitespace
  getline(cin,str);

  // Compare two strings
  if(str.compare(str2) != 0)
  cout << "Not equivalent" << endl;

  // Find length of string
  int n = str.length();

  // Substrings
  string str3 = str2.substr(2,5);

  return 0;
}
```