C++ Programming

The Basics

The Basics

- Statements and expressions
- Control flow
- Structures

Statements

- As in C, a statement in C++ is a command in a program to direct the program to take a particular action
- Likely the simplest statement in C++ is an expression statement
  - This is simply an expression (which is optional) followed by a semicolon(;)
  - This does mean that the following (the null statement) is a valid statement:

```
;
```

- Usually though, an expression is given and that expression is evaluated when the statement is executed

Expressions and Types

- Every expression has a type that determines the operations that may be performed on it
- A declaration is a statement that introduces a name into a program, specifying a type for the named entity
- For example:

```
int foo;
```

introduces a named variable foo that is of type integer

Expressions and Types

- C++ has a similar set of basic types to C
- This includes things like bool, char, int, and double
- Each fundamental type corresponds directly to hardware facilities and has a fixed size that dictates the range of values that it can represent
- C++ also has a collection of other types that C traditionally does not; we will touch on some of those later on

Expressions and Types

- C++11 introduced the concept of auto typing
- In this, the keyword auto can be used to avoid explicitly declaring the type of an entity (like a variable)
- Instead, the type is inferred by the context, such as the type of the expression used to initialize a variable
- For example:

```
auto n = 0;     // Create an integer variable n and initialize it to 0
```

Expressions and Types

- The use of auto in C++ is divisive in the community, however
  - Purists say it should be avoided and types should always be explicitly declared for readability and to maintain compatibility with older C++ code and compilers
  - Others, however, find the shorthand useful and a step towards languages without explicit declarations (like Python)
- Either way, you will see auto in the wild, so it is good to know what it does and how to use it

Constant Expressions

- C++ has multiple concepts of constants:
  - `#define` constants – preprocessor definitions that do a simple replacement during preprocessing and before compiling
  - `const` constants – named constant declarations where the programmer commits to not changing a value, and this promise is enforced by the compiler
  - `constexpr` constants – constant expressions evaluated at compile time, allowing them to be placed in read-only memory to improve performance (new in C++11)

Constant Expressions

```
#define PI 3.14
constexpr double circleArea(double x) { return PI*x*x; }

int main() {
    const double pi = 3.14;
    constexpr double radius1 = 5.0;
    const double radius2 = 10.0;

    constexpr double area1 = circleArea(radius1);
    constexpr double area2_error = circleArea(radius2);
    const double area2_good = circleArea(radius2);
}
```

Expressions and Operators

- C++ also uses a similar set of operators to C
- Arithmetic operators like +, -, *, /, and %
- Comparison operators like ==, !=, <, >, <=, and >=
- Assignment and initialization operator as =
- In assignments and arithmetic operations, C++ does do conversions between basic types as in C

Expressions and Operators

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

int main()
{
   int sum;
   bool bigEnough;
   sum = 50 + 50;
   bigEnough = sum >= 100;
   cout << "Sum is " << sum << ". Is it big
     enough?  " << bigEnough << endl;
}
```

Statements and Blocks

- A block, or compound statement, is the term given to a collection of statements enclosed in {…}
  - This is the way that a program can group multiple statements together into a single statement, often for the purposes of control flow
  - A block also defines a scope for variables declared in it; local variables are put on the stack for the duration of the execution of the block statement
  - Such variable declarations are optional, and no new variables need be introduced in a block
  - Originally all such declarations had to be at the beginning of the block, but they can now generally be interwoven with statements through the code

Control Flow

- In C++, the flow of control in a program behaves very similarly to how things are done in C
  - Sequence: The normal stepping through from one statement to the next
  - Selection: To enable selection for the next statement or block from amongst a number of possibilities
  - Repetition: Repetition of a single statement or of a block; repetition can take many forms depending on the needs of the program, including for(...), while(...), and do...while loops.

Control Flow – Selection

- One-way selection is accomplished using a simple if

```
if (<logical expression>)
     <statement1> //performed if expression is true
```

Control Flow – Selection

- Two-way selection is accomplished using if-else

```
if (<logical expression>)
     <statement1> //performed if expression is true
else
     <statement2> //performed if false
```

Control Flow – Selection

- Multi-way selection using the else if

```
if(<expression1>)
   <statement1>
else if (<expression2>)
   <statement2>
else if (<expression3>)
   <statement3>
else
   <statement4> // otherwise -- default
```

Control Flow – Selection

- Multi-way selection using the switch / case statement

```
switch (<expression>){
case <constant-exp1>: <statement(s)1> [break;]
case <constant-exp2>: <statement(s)2> [break;]
case <constant-exp3>: <statement(s)3> [break;]
default: <statement(s)d>; [break;]
}
```

Control Flow – Repetition

- while loops:

```
while(<expr>) <statement>
```

```
int count = 0;
while (count < 10) {
     cout << "Count is: " << count << endl;
     count++;
}
```

Control Flow – Repetition

- for loops:

```
for(<expr-init>;<condition expr>;<expr-iter>)
     <statement>
```

```
for(int count = 0; count < 10; count++) {
     cout << "Count is: " << count << endl;
}
```

Control Flow – Repetition

- do-while loops:

```
do
     <statement>
while (<expr>);
```

```
int count = 0;
do {
     cout << "Count is: " << count << endl;
     count++;
} while (count < 10);
```

Structures

- In C++, a user-defined record (aggregate type) is called a structure and is referred to as a `struct` in a program
- They are handled in much the same way as they are in C, with slight differences (that ultimately make them easier to use and refer to)
- Interestingly in C++, structures and classes are more-or-less equivalent except for their default visibility:
  - In a structure, members default to public; whereas,
  - In a class, members default to private.
- Much more on classes shortly!

Structures

- As noted previously, structures are defined using the `struct` keyword

```
struct Point {
  int x;
  int y;
};
Point p1, p2;
```

- Notice the difference between how C++ and C would handle declaring things using our new `Point` structure?

Structures

- C would require us to refer to our structure as `struct Point` or else it would generate an error during compilation
- As this would get tedious, you could use the `typedef` directive to name a new type to avoid having to do this
- In C++, however, this is not necessary; you can do things the C way, but you are not required to do so

Structures

- Accessing fields: Individual fields of a structure can be accessed using the "." syntax

```
Point p1;
p1.x = 3;
p1.y = 2;
```

When we have a pointer to a structure (more on pointers in a minute), we can instead access fields using "->"

Structures

- Assignment: A structure can be assigned as a complete unit:

```
Point p1, p2;
p1.x = 3;
p1.y = 2;
p2 = p1;
```

This last statement is equivalent to:

```
p2.x = p1.x;
p2.y = p1.y;
```