# Lecture 7

## Namespaces, Variable scopes, control structures, and functions

Instructor: Ashley Gannon

ISC3313 Fall 2021

# Namespaces

# Namespaces

There are three main namespaces used in C++:

1. `std`: All library entities are defined within namespace `std`. This includes namespaces nested within namespace `std`.
2. `abi`: Specifies a number of type and function.
3. `__gnu_`: Indicating one of several GNU extensions. Choices include `__gnu_cxx`, `__gnu_debug`, `__gnu_parallel`, and `__gnu_pbds`.

We'll look more closely at `std`.

# Namespace `std` for `iostream`

`iostream` is a header that defines the standard input/output stream objects:

**`cin`**

Standard input stream

**`cout`**

Standard output stream

**`cerr`**

Standard error stream

**`clog`**

Standard output stream for logging

As noted in the last slide, all library components of `iostream` are defined in the namespace `std`. In order to use the objects in `iostream`, you must do one of two things:

1. Use a *using* declaration in your header file.
   i.e. `using namespace std`.
2. Use a fully qualified library name for each call.
   i.e `std::cout` or `std::cin`

# Revisiting "Hello world!" using `iostream`

**Option 1:**

```
#include <iostream>

using namespace std;

int main()
{
cout«"Hello world!"«endl;
}
```

**Option 2:**

```
#include <iostream>


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

# Variable scope

# What is the scope of a variable?

We've seen how to declare different types of data, or variables. Are variables automatically recognized in every part of the program?

```
#include <iostream>

using namespace std;
int main()
{
    bool a = true;
    if (a)
    {
        cout « a « endl;
    }
}
```

In this case the variable `a` is obviously known inside the conditional statement.

# What is the scope of a variable

In the following case, is the variable `b` known outside of the conditional statement?

```
#include <iostream>

using namespace std;
int main()
{
    bool a = true;
    if (a)
    {
        double b = 15.0;
    }
    cout « b « endl;
}
```

# What is the scope of a variable

Initialization should be done in the same scope that the variable is needed:

```
#include <iostream>

using namespace std;
int main()
{
    bool a = true;
    double b;
    if (a)
    {
        b = 15.0;
    }
    cout « b « endl;
}
```

# Local variables versus global variables

This brings up the concept of variable scope, and local versus global variables.

- A local variable lives within the block of code, denoted by `{ }` symbols, that it is declared in. It will also exist in any new block that is declared within the original block (like the first example).
- A global variable is known to every part of the program.

# Declaring global variables

Global variables can be declared outside of the `main()` function:

```
double b = 15.0;
int main()
{
    cout « b « endl;
}
```

This gives us 15.0 as output.

# Declaring global variables

They can be modifed as well. What does the following program output?

```
double b = 15.0;
int main()
{
    cout « b « endl;
    b = 10;
    cout « b « endl;
}
```

# Declaring global variables

They can be modifed as well. What does the following program output?

```
double b = 15.0;
int main()
{
    cout « b « endl;
    b = 10;
    cout « b « endl;
}
```

This gives us 15.0 as output, followed by 10.0.

# Control structures

# If statements

We've seen how a basic `if` statement works, but we can add more than one condition:

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

# While loop

The while loop executes a block of code until some condition is satisfied:

```
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

If the condition is not true initially, the loop will not execute at all.

# Do-while loop

This type of while loop *always executes at least once*:

```
// number echoer

#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

This is useful if the condition itself is not known until the block of code executes.

# For loop

Formally, the pattern of a `for` loop is:

```
for (initialization; condition; increase) statement;
```

The procedure is as follows:

1. `initialization` is executed once at the beginning. This is typically used to set a counter to 0, or the beginning of an arrary.
2. `condition` (which is expecting a bool) is checked. If the condition is true, the loop executes at least once.
3. `statement` is executed. This can be a single line of code, or many lines of code enclosed in a block `{}`.
4. Whatever code is specified in the `increase` field is executed. This can be anything.

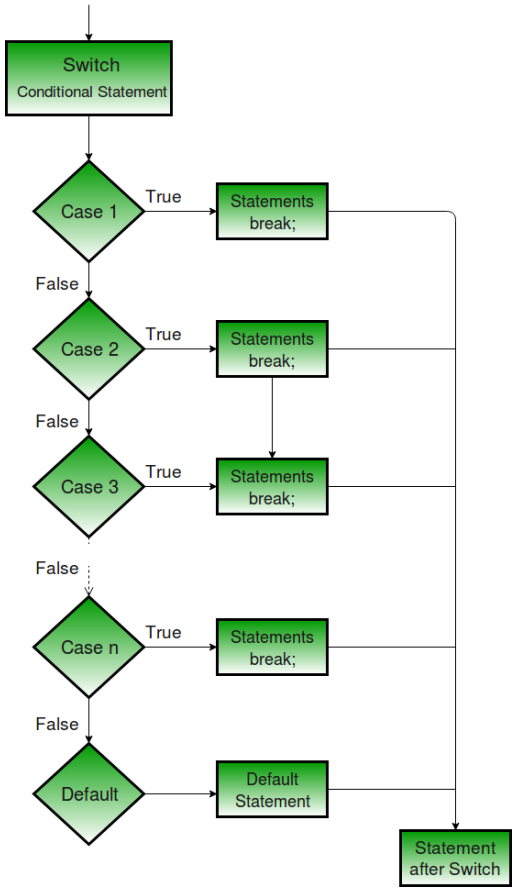
# Jump statements

There are some special keywords related to loops that are useful:

- `break` causes the loop to be exited for any reason
- `continue` causes the remainder of a code block to be ignored, and the `increase` field to be executed and so on...
- `goto` skips directly to a specified point in the code. There is almost never a good reason to use this.
- `exit` terminates the entire code execution, not just a loop.

# Switch statement

A `switch` statement is used to allow the code execution to choose between a number of options, similar to `if-else` statements.



# Switch statement

What happens if there are not `break` statements at the end of each `case`?

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

# Functions

# Basics of functions

Functions are like a block of code with additional features:

- functions return a value, unless `void` used
- can return any data type
- arguments are passed to the function, these can be of any data type
- arguments can be modifed within the function, or not

Their structure looks like:

```
type name (param1, param2, ...)  { statements }
```

# Calling functions

Functions must be *declared* before the `main` program executes. It can be *defined* anywhere.

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

```
The result is 8
```

Could also just have header at the top: `int addition(int, int);` and have the full definition below the `main` function.

# Void functions

Functions can have a `void` return type and return nothing.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

```
I'm a function!
```

# Arguments

Functions can have arguments of any data type. When a function receives an argument, it makes its own copy of the data. This is known as *pass-by-value*. Changing the value within the function has no effect on the value outside of the function.

# References

A reference variable, initialized with an ampersand, becomes an alternative name for a given variable. It has access to the same address as the original in memory.

```
double b = 15.0;
double &c = b;
c = 10.0;
cout « b « endl;
b = 11.0;
cout « c « endl;
```