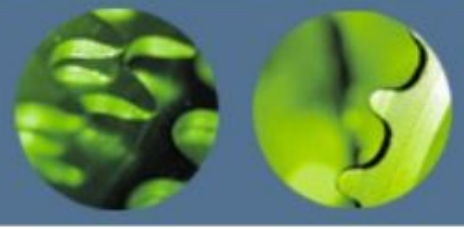
SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College Of Engineering

C++ Basics (Module 1.3)

Dr. Ayesha Hakim

What is C++?

- The C++ programming language can be thought of as a "**superset**" of the C language: a legal ANSI C program is a legal C++ program and means the same thing.



Brief History of C++

- Derived from C
  - Adds classes and other features
  - Including the increment operator, ++
- Developed by Bjarne Stroustrup at Bell Labs in 1979
- Standardized by ANSI-ISO in 1998

Why study C++ ?

1. Foundational Knowledge
2. System Programming
3. Performance-Critical Applications
4. Legacy Codebases
5. Game Development
6. Embedded Systems
7. Competitive Programming
8. Cross-Platform Development
9. Performance Critical Libraries
10. Understanding Memory Management

A simple C++ program

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

//This program displays a "HellO Word" message
int main( ){
    cout<<"Hello Word"<<endl;
    return 0;
    }
```

- Save as hello.cpp

Explanation

- Files that contain C++ programs are named using the .cpp file extension (for example hello.cpp).
- The statement `#include<iostream>` is a preprocessor directive that is included in C++ program to allow the program to perform the input and output.
- `iostream` is a standard C++ header file that defines `cin`, `cout`, and the operators << and >>.
- `using namespace std` is written to avoid prefixing standard library names like `cin, cout, endl` with `std::`
- All the elements of the standard C++ library are declared within what is called a `namespace`, the namespace with the name std. So in order to access its functionality we declare with this expression that we will be using these entities.
- Files that contains C++ programs must have a `main()` function to indicate where the program execution begins.
- The `main( )` function usually is written to return an integer value.
- The statement `return 0` will tell C++ to exit the `main()` function.

Hello, World! explained

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

The *main* routine – the start of **every** C++ program! It returns an integer value to the operating system and (in this case) takes no arguments: main()

The **return** statement returns an integer value to the operating system after completion. 0 means "no error". C++ programs **must** return an integer value.

Hello, World! explained

loads a *header* file containing function and class definitions

Loads a *namespace* called *std*. Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line in this tutorial.

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

- *cout* is the *object* that writes to the stdout device, i.e. the console window.
- It is part of the C++ standard library.
- Without the "using namespace std;" line this would have been called as *std::cout*. It is defined in the *iostream* header file.
- << is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left. *endl* is the C++ newline character.

What is `#include <iostream>` ?

- #include tells the precompiler to include a file
- Usually, we include header files
  - Contain *declarations* of structs, classes, functions
- Sometimes we include template *definitions*
  - Varies from compiler to compiler
  - Advanced topic we'll cover later in the semester
- <iostream> is the C++ label for a standard header file for input and output streams

What is `using namespace std;` ?

- The **using** directive tells the compiler to include code from libraries that have separate *namespaces*
  - Similar idea to “packages” in other languages
- C++ provides a namespace for its standard library
  - Called the “standard namespace” (written as std)
  - cout, cin, and cerr standard iostreams, and much more
- Namespaces reduce collisions between symbols
  - Rely on the `::` scoping operator to match symbols to them
  - If another library with namespace mylib defined **`cout`** we could say **`std::cout vs. mylib::cout`**
- Can also apply **`using`** more selectively:
  - *E.g.,* just **`using std::cout`**

What Is the C++ Namespace?

- Namespace in C++ is the declarative part where the scope of identifiers like functions, the name of types, classes, variables, etc., are declared. The code generally has multiple libraries, and the namespace helps in avoiding the ambiguity that may occur when two identifiers have the same name.
- **Example**, suppose you have two functions named calculate(), and both are performing different tasks. One calculate() function is doing the multiplication, and another is doing the addition. So in this case, to avoid ambiguity, you will declare both the functions in two different namespaces. These namespaces will differentiate both the functions and also provide information regarding both the functions.

Important points to consider while declaring a namespace:

- You can only define them in a global scope.
- It is only present in C++ and not in C.
- To access a class inside a namespace, you can use namespacename::classname.

Example:

```
#include <iostream>

namespace column
{
  int data = 20;
}

int main()
 {
  float data= 140.57;

  std::cout<<" Variable = "<<data<<std::endl;

  std::cout<<" Variable declared in namespace = "<<column::data;

  return 0;
 }
```

Same name

Here data variable whose value is 20 is declared in the namespace and another variable data having value 140.57 is declared in the main function. To print both the variables we have to put the namespace name in front of the data variable having value 20 i.e column::data, otherwise both data variables will print 140.57.

So to print the value of a namespace member, we have to write a namespace with its scope, which will avoid the name clash.

```
1  // Fig. 1.2: fig01_02.cpp
2  // A first program in C++
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome to C++!\n";
8
9     return 0;      // indicate that progr
10 }
```

```
Welcome to C++!
```

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with # are preprocessor directives.

`#include <iostream>` tells the preprocessor to

C++ programs contain one or more functions, one of which must be `main`

Parenthesis are used to indicate a function

`int` means that `main` "returns" an integer value.
More in the next class.

```
1  // Fig. 1.2: fig01_02.cpp
2  // A first program in C++
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome to C++!\n";
8
9     return 0;      // indicate that progr
10 }
```

```
Welcome to C++!
```

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with # are preprocessor directives.

`#include <iostream>` tells the preprocessor to include the contents of the file `<iostream>`, which includes input/output operations (such as printing to the screen).

A left brace { begins the body of every function and a right brace } ends it.

```
1  // Fig. 1.2: fig01_02.cpp
2  // A first program in C++
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome to C++!\n";
8
9     return 0;      // indicate that progr
10 }
```

```
Welcome to C++!
```

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with # are preprocessor directives.

`#include <iostream>` tells the preprocessor to include the contents of the file `<iostream>`, which includes input/output operations (such as printing to the screen).

Prints the *string* of characters contained between the quotation marks.

The entire line, including `std::cout`, the `<<` *operator*, the *string* `"Welcome to C++!\n"` and the *semicolon* (`;`), is called a *statement*.

All statements must end with a semicolon.

very function

```
1  // Fig. 1.2: fig01_02.cpp
2  // A first program in C++
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome to C++!\n";
8
9     return 0;      // indicate that progr
10 }
```

```
Welcome to C++!
```

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with # are preprocessor directives.

`#include <iostream>` tells the preprocessor to include the contents of the file `<iostream>`, which includes input/output operations (such as printing to the screen).

Prints the *string* of characters contained between the

ncluding `std::cout`, the `<<`
ing `"Welcome to C++!\n"` and
), is called a *statement*.

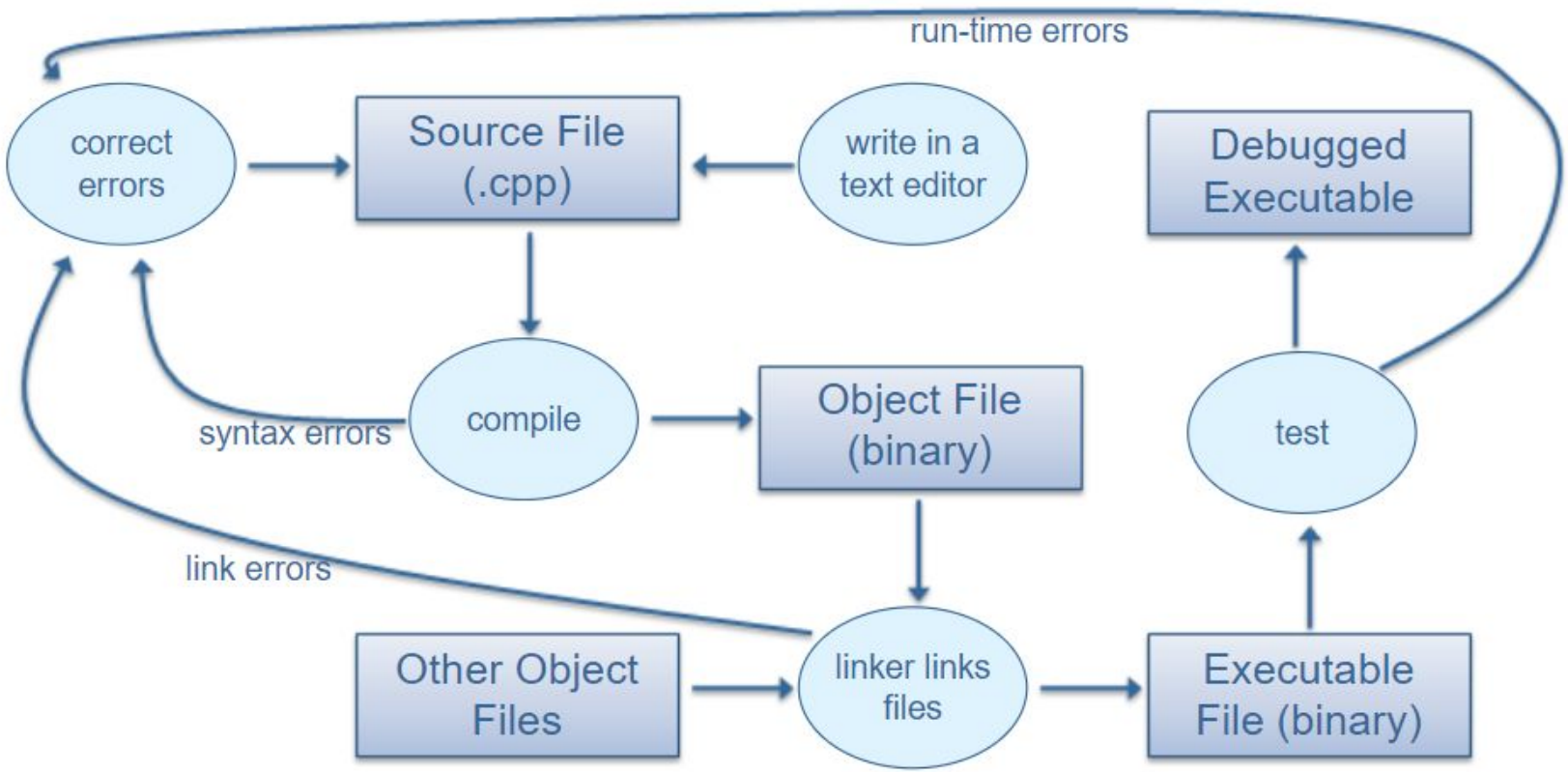
All statements must end with a semicolon.

very function

`return` is a way to exit a function from a function.

`return 0`, in this case, means that the program terminated normally.

Processing a C++ Project

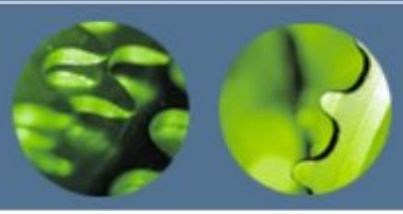


Header Files

- C++ (along with C) uses *header files* as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.

C++ language headers aren't referred to with the .h suffix. <iostream> provides definitions for I/O functions, including the *cout* function.

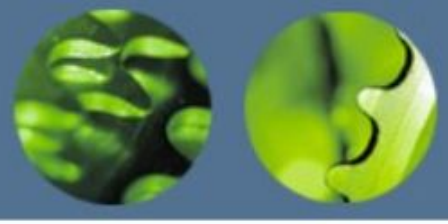
```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```



Including Files

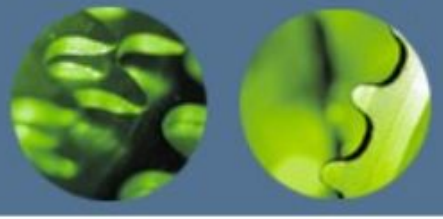
- The #include directive is used to insert a library file into a program
  - The library file contains declarations (names) of all of the functions in the library
- There are two general forms of the #include directive
  - #include <library>
    - The file is part of the C++ language
  - #include "library"
    - The file location should be specified by the programmer

```
#include <iostream>
using namespace std;
int main(){
     cout << "Hello World!";
     return 0;
}
```



Preprocesser Directives

- The # indicates a command for the C++ preprocessor
  - Known as a preprocessor directive
  - Such commands do not end in a semi-colon
- The C++ preprocessor copies the contents of the included file into the program
  - Replacing the line containing the directive
  - Such files contain function headers



User Libraries

- Unlike standard libraries the preprocessor needs the location of user libraries
  - Such libraries are given in quotes ("")
  - The location of the file must be specified by the programmer
  - Locations are given relative to the directory containing the source file
  - File extensions should be given
    - e.g. #include "myfile.h"

Primitive data types, sizes, limits

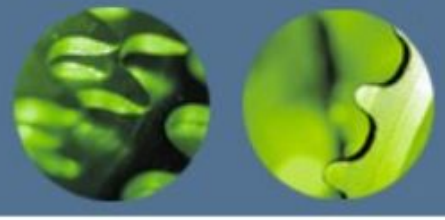
Type	Size (bytes)	Min Value	Max Value
bool	1	0	1
char	1	-128	127
signed char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	4	-2147483648	2147483647
unsigned long	4	0	4294967295
long long	8	-9223372036854775808	9223372036854775807
unsigned long long	8	0	18446744073709551615
float	4	1.17549e-38	3.40282e+38
double	8	2.22507e-308	1.79769e+308
long double	16	3.3621e-4932	1.18973e+4932

Explanation of cin and cout w.r.t. iodemo.cpp

- The expression cin >> a causes the program to read an integer into the variable a, from the standard input.
- Similarly, cin >> b reads an integer into b, but cin >> c interprets its input as a real number, and stores it in c.
- The next statement, cin >> str reads characters until a whitespace is encountered, and stores a null-terminated string in str (excluding the whitespace).
- Note that if you typed a very long string (i.e., more than 20 characters), your program could crash.
- To read a string with spaces, use getline(cin, str);

OOP Basics in C++

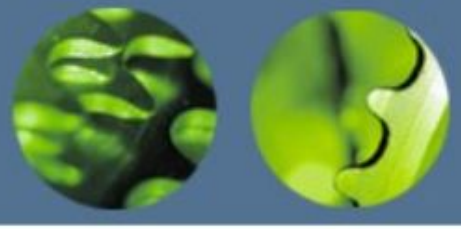
- `class` Keyword
  - The class keyword is used to define a new class.
  - A class is a blueprint for creating objects, which can include both data members (variables) and methods (functions)
- By default, all members of a class are **private** if no access specifier is declared.
- This means they can only be accessed and modified by member functions of the class and not from outside the class.
- The public keyword makes the members accessible from outside the class.



Main Function

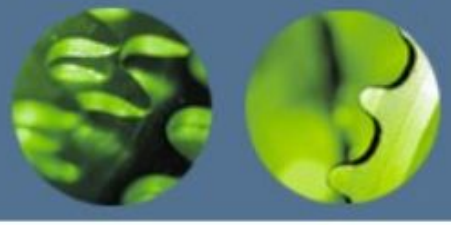
- A C++ program is made up of a number of functions
  - A function is self contained part of a program
  - Every C++ application has a main function
  - The main function is called when the executable file is run
- The main function is made up of two parts
  - Its header: int main()
  - Its body: the code in {}s

```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World!";
    return 0;
}
```



More About {}s

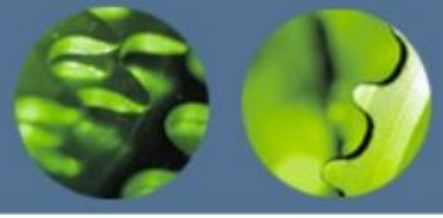
- The main function's body starts with an opening {
  - And ends with a closing }
- These curly brackets are used to indicate a body of code
  - Belonging to a function, or
  - Loop, or
  - Decision



Screen Output

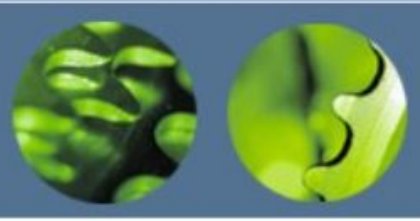
- cout is a function in the iostream library
  - It outputs to the screen
  - The << operator is often called the insertion operator
- "Hello World!" is a string
  - Words contained in double quotes ("") are treated as arbitrary text by the compiler
  - not program instructions

```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World!";
    return 0;
}
```



Semi-Colons

- A semi-colon indicates the end of a C++ command
  - By convention most C++ commands are written on one line
  - However, the newline character does not indicate the end of a command
- Don't forget semi-colons
  - Omitting them usually prevents the program from compiling



Return Statement

- The return statement ends a function
  - More correctly, ends the function's invocation
  - It does not have to be the last line of a function
    - But typically is the last line of a **main** function
- The type of the returned value should match the function's return type
  - In this case 0 is an integer (or int)
  - More on this later ...

```
#include <iostream>
using namespace std;
int main(){
     cout << "Hello World!";
     return 0;
}
```

Why Use Access Specifiers

- **Encapsulation**
  - Encapsulation is one of the fundamental principles of object-oriented programming (OOP).
  - By making member variables private, we control how they are accessed and modified.
  - This helps in maintaining the integrity of the data.
- **Interface vs. Implementation**
  - Public methods serve as the interface through which the outside world interacts with the class, while private members and methods are the implementation details that are hidden from the outside world.

Public Section

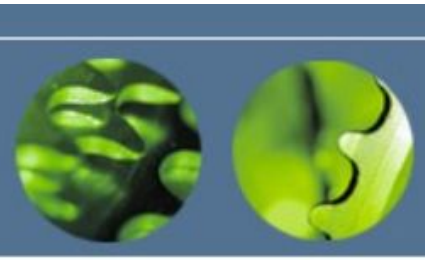
- Members declared in the public section can be accessed from outside the class.
- This is where you usually put the interface of the class: methods and constructors that users need to access.

```
class MyClass {
public:
    void publicMethod() {
        // Method accessible from outside the class
    }
};
```

Private Section

- Members declared in the private section can only be accessed by other members of the class.
- This is used to encapsulate the internal state and implementation details of the class.

```
class MyClass {
private:
    int privateMember;
    void privateMethod() {
        // Method not accessible from outside the class
    }
};
```



Summary

- Include libraries required for your program
  - The iostream library is required for standard input and output
  - Specify the std namespace
- Write a main function
  - That implements the solution's algorithm
  - The body of the function is contained in {}s
  - Statements in function bodies end with a ;

Constructors in C++

- A constructor is a special member function of a class that is automatically called when an object of that class is created.
- Constructors are used to initialize the objects of a class.
- They have the same name as the class and do not have a return type.

Types of Constructors

- Default Constructor
  - A constructor that takes no arguments.
  - It is called when an object is created without any parameters.
  - Example:

```
class MyClass {
public:
    MyClass() {
        // Default constructor body
    }
};
```

- Parameterized Constructor:
  - A constructor that takes one or more arguments.
  - It is used to initialize objects with specific values.
  - Example:

```
class MyClass {
public:
    MyClass(int value) {
        // Parameterized constructor body
    }
};
```

- Member Initialization List
  - A member initialization list is a more efficient way to initialize class members.
  - It is preferred over assigning values in the constructor body, especially for initializing const members and reference members.
  - Example:

```
class MyClass {
    int x;
    const int y;
    float z;
public:
    MyClass(int a, int b, float c) : x(a), y(b), z(c) {
        // Constructor body (if needed)
    }
};
```

Creating Objects in C++

- Using the Default Constructor
  - Syntax: classname objectname;
  - Example:

```
int main(){
    MyClass m1;
}
```

- Using a Parameterized Constructor
  - Syntax: className objectName(arguments);
  - Example:

```
int main(){
    MyClass m2(3,4,5.5);
}
```

Refer classDemo.cpp

Integer and Floating-point Division

- The behavior of division operator (/) depends on the type of the operands.
- If both operands are integers, C++ performs integer division. That means any fractional part of the answer is discarded, making the result an integer.
- If one or both operands are floating-point values, the fractional part is kept, making the result floating-point.

Type Conversion in C++

- C++ operations occur on same type operands. If operands are of different types, C++ will convert one.
- Implicit conversion do not require any operator. They are automatically performed.

→ No data loss →		
Lower Data Type	char → short → int → long → float → double → long double	Higher Data Type
← Data loss ←		

Explicit Type Conversion (Casting)

- Syntax of the explicit type casting
- (type) expression;
- Syntax of the Static Cast
- static_cast < new_data_type> (expression);

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as '**automatic type conversion**'.
    - Done by the compiler on its own, without any external trigger from the user.
    - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.

All the data types of the variables are upgraded to the data type of the variable with largest data type.
bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion

```
// An example of implicit conversion

#include <iostream>
using namespace std;

int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c

        // y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;

        // x is implicitly converted to float
        float z = x + 1.0;

        cout << "x = " << x << endl
                << "y = " << y << endl
                << "z = " << z << endl;

        return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

Explicit Type Conversion

This process is also called **type casting** and it is user-defined. Here the user can typecast the result to make it of a particular data type.
In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

  Syntax:

  ```
  (type) expression
  ```

  where *type* indicates the data type to which the final result is converted.

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
using namespace std;

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    cout << "Sum = " << sum;

    return 0;
}
```

Output:

```
Sum = 2
```

Conversion using Cast operator: A Cast operator is an **unary operator** which forces one data type to be converted into another data type.

C++ supports four types of casting:

1. Static Cast
2. Dynamic Cast
3. Const Cast
4. Reinterpret Cast

Example:

```
#include <iostream>
using namespace std;
int main()
{
    float f = 3.5;

    // using cast operator
    int b = static_cast<int>(f);

    cout << b;
}
```

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

Output:

```
3
```

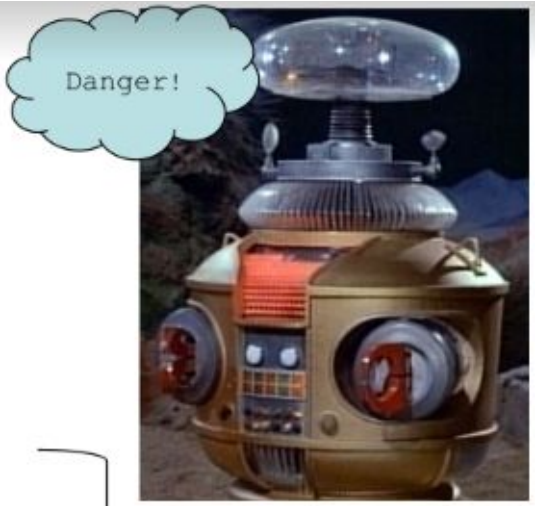
Type Casting

- `static_cast<new type>( expression )`
  - This is exactly equivalent to the C style cast.
  - This identifies a cast **at compile time**.
  - This will allow casts that reduce precision (ex. double □ float)
  - ~99% of all your casts in C++ will be of this type.

```
double d = 1234.56 ;
float f = static_cast<float>(d) ;
// same as
float g = (float) d ;
```

- `dynamic_cast<new type>( expression)`
  - Special version where type casting is performed at runtime, only works on reference or pointer type variables.
  - Usually handled automatically by the compiler where needed, rarely done by the programmer.

Type Casting cont'd



- `const_cast<new type>( expression )`
  - Variables labeled as *const* can't have their value changed.
  - const_cast lets the programmer remove or add *const* to reference or pointer type variables.
  - If you need to do this, you probably want to re-think your code.
- `reinterpret_cast<new type>( expression )`
  - Takes the bits in the expression and re-uses them **unconverted** as a new type. Also only works on reference or pointer type variables.
  - Sometimes useful when reading in binary files and extracting parameters.

"unsafe": the compiler will not protect you here!

The programmer must make sure everything is correct!

Control Flow statements

- The syntax of if-else, switch-case, while loop, do-while loop, for loop, break, continue is exactly the same as you have learned in C.

Questions?