C++ Programming basics

print & read functionality defines in iostream or std namespace?

Why we use both?

by default main() return in C & C++?

Dr. Alekha Kumar Mishra

C++

- C++ is developed by Bjarne Stroustrup in 1980
- C++ is an extension to C
- C++ is a superset of C
- Inherit all ANSI C directives and C functions
- The most important elements added to C to create C++ concern classes, objects, and object-oriented programming.
- C++ programmers take advantage of the rich collections of classes and functions in the C++ Standard Library
- C++ has many other new features as well, including an improved approach to input/output (I/O).
- C++ programs are fast and efficient.

A simple C++ program

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {
    cout << “Every age has a language of its own\n”;
    return 0;
}
```

Is the following program correct?

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

int main () { cout
<<
“Every age has a language of its own\n”
; return
0;}
```

Answer

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

int main () { cout
<<
“Every age has a language of its own\n”
; return
0;}
```

- C++ compilers ignores two or more number of space characters or new line feed.

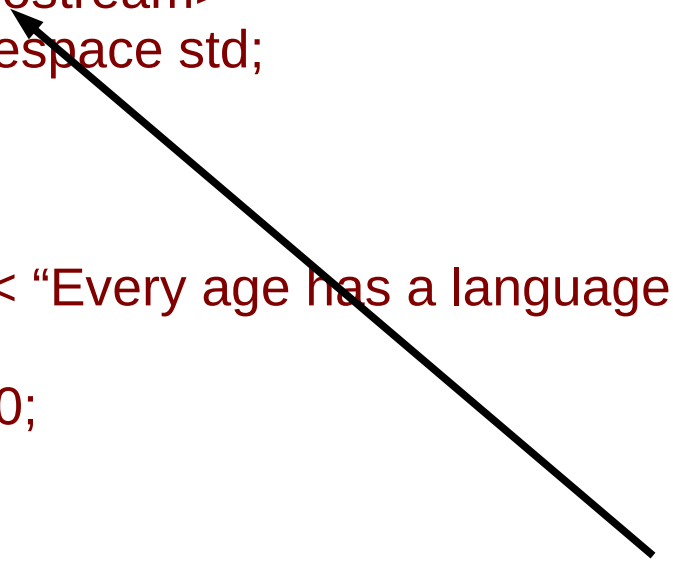
Analyzing the simple program

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

    cout << “Every age has a language of its own\n”;

    return 0;
}
```



- The preprocessor directive #include tells the compiler to insert another file into your source file. In effect, the #include directive is replaced by the contents of the file indicated.
- Here, includes the contents of the input/output stream header <iostream>
- .h extension is not used for C++ header files

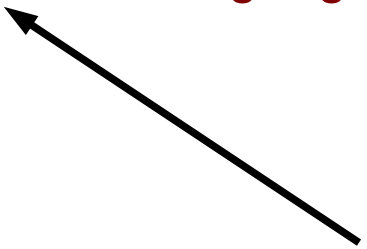
Analyzing the simple program

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

    cout << “Every age has a language of its own\n”;

    return 0;
}
```



- The identifier cout is actually an object predefined in C++, corresponds to standard output stream
- The operator << directs the contents of the variable on its right to the object on its left.
- It directs the string constant (the output statement) to cout, which sends it to the display device.
- In C++, operators << is overloaded for “**put to**” operation.

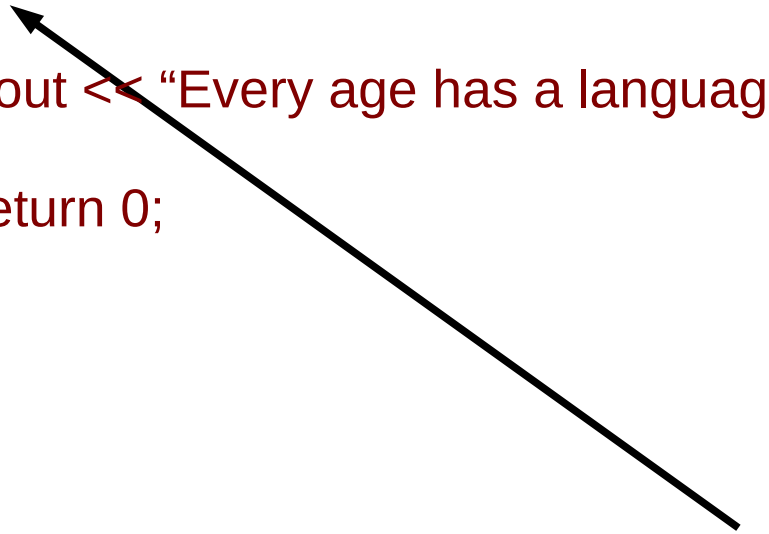
Analyzing the simple program(2)

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

      cout << “Every age has a language of its own\n”;

      return 0;
}
```



- What is the importance of main() function ???

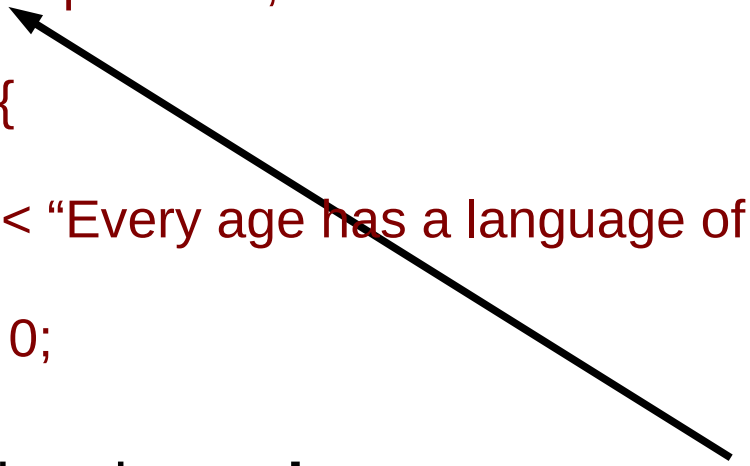
Analyzing the simple program(4)

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

     cout << “Every age has a language of its own\n”;

     return 0;
}
```



- The directive **using namespace std;** says that the program that follow uses the **std** namespace.
- Various program components such as **cout** object are declared within this namespace.
- If we didn’t use the using directive, we would need to add the std name to them.
  - std::cout << “Every age has a language of its own.”;

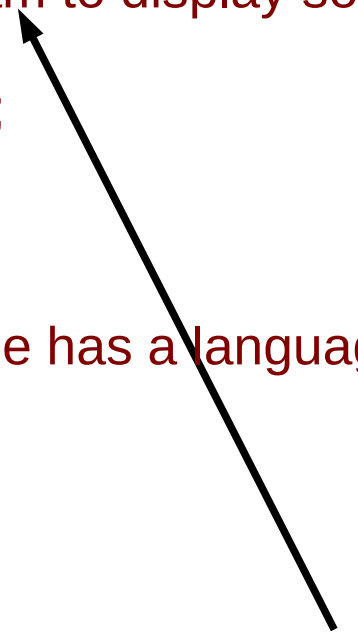
Analyzing the simple program(5)

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

      cout << “Every age has a language of its own\n”;

      return 0;
}
```



- A Single line comment!!!
- We can also use C's multiline comment in C++

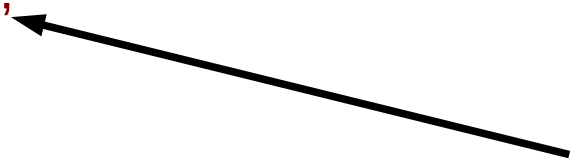
Analyzing the simple program(6)

```
// A simple c++ program to display some text
#include <iostream>
using namespace std;

int main() {

    cout << “Every age has a language of its own\n”;

    return 0;
}
```



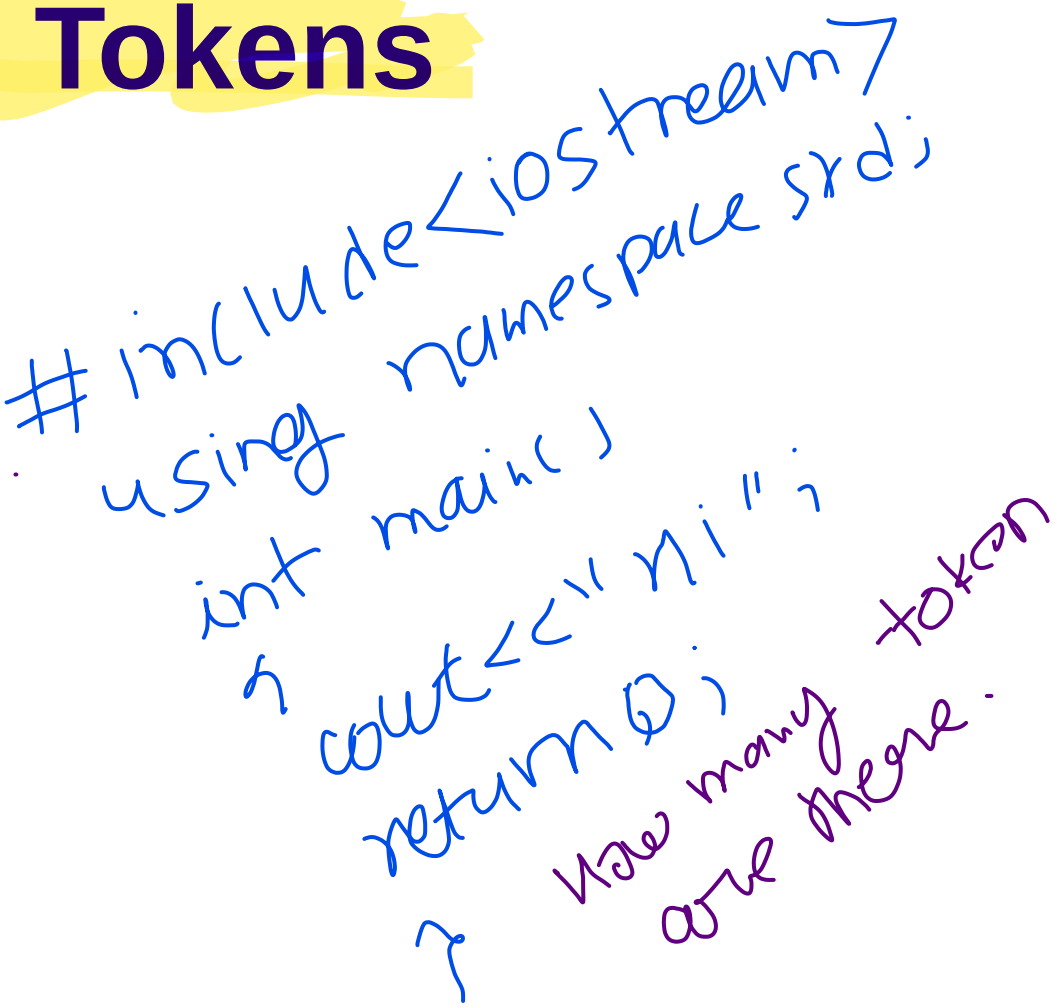
- Why return a 0 in main function ???

How to Write and Execute in Linux/Unix Environment

- Open a file in vi or any other text Editor with extension as .cpp or .cxx or .C
- Write the source code
- Save and exit
- Compile with g++ <filename> (or c++ <filename>)
- Check for errors
- Execute with ./a.out
- Guideline:- Do write the source code in well indented format

C++ Tokens

- Keywords
- Identifiers
- Constants
- Strings
- Operators



Keywords

asm ←	dynamic_cast ←	new ←	template ←
auto	else	operator ←	this ←
bool ←	enum	private ←	throw ←
break	extern	protected ←	true ←
case	false ←	public ←	try ←
catch ←	float	register	typedef
char	for	reinterpret_cast ←	typeid ←
class ←	friend ←	return	union
const	goto	short	unsigned
const_cast ←	if	signed	using ←
continue	inline ←	sizeof	virtual ←
default	int	static	void
delete ←	long	static_cast ←	volatile
do	mutable ←	struct	wchar_t ←
double	namespace ←	switch	while

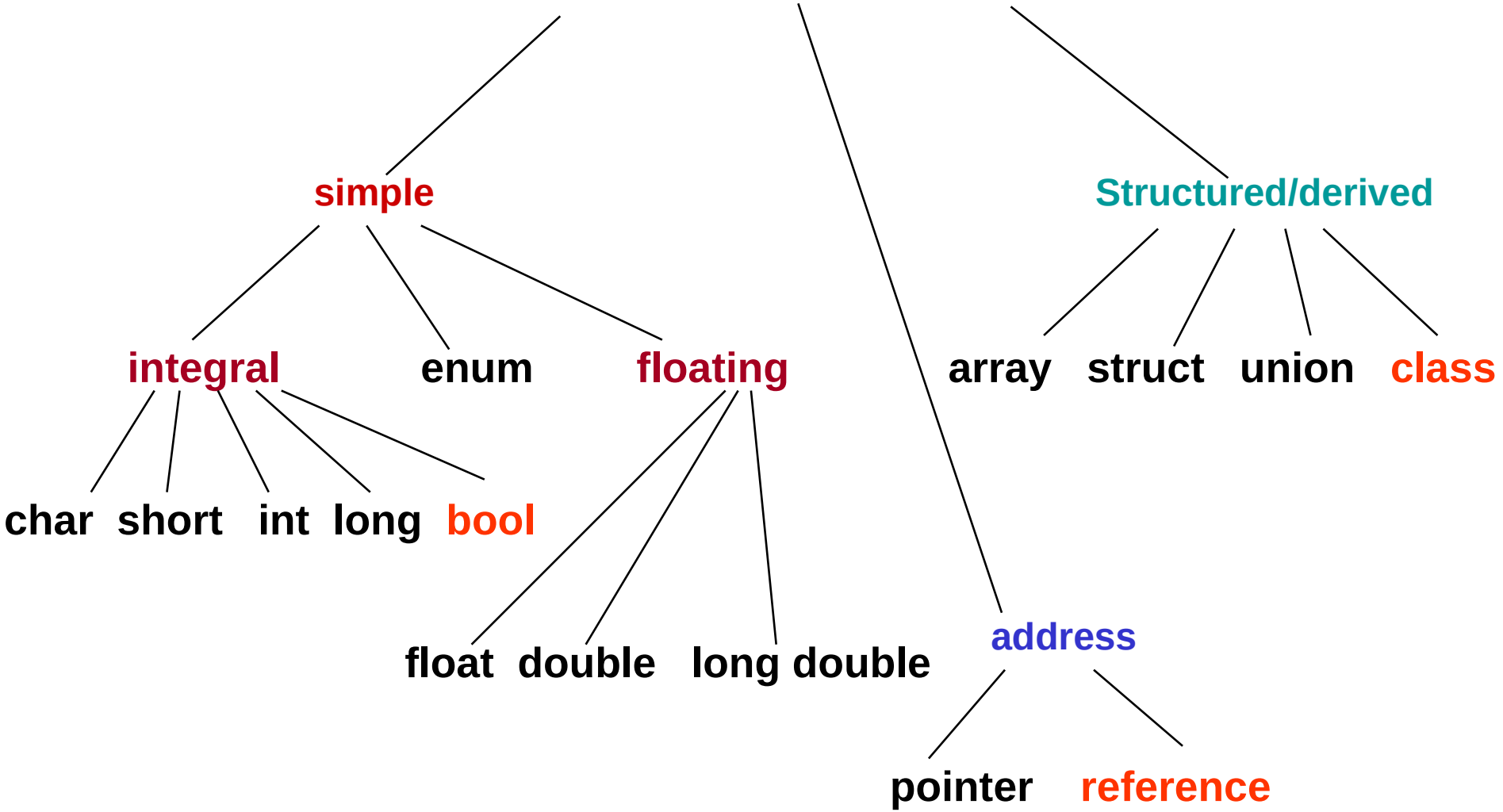
Identifiers

- The names given to variables (and other program features) are called identifiers.
- Rules for writing identifiers
  - We can use upper- and lowercase letters, underscore, and the digits from 1 to 9. 0 is included or not?
  - The first character must be a letter or underscore.
  - Identifiers can be as long as you like, but most compilers will only recognize the first few hundred characters.
  - The compiler distinguishes between upper and lowercase letters, so Var is not the same as var or VAR.

Rules for identifiers contd..

- You can't use a C++ keyword as a variable name.
- Many C++ programmers follow the convention of using all lowercase letters for variable names.
- Names in all uppercase are sometimes reserved for constants
- A variable's name should make clear to anyone reading the listing the variable's purpose and how it is used.
  - Thus, boilerTemperature is better than something cryptic like bT or t.

C++ data types


simple
Structured/derived
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

Integers

- Exist in several sizes
- The most commonly used is type int.
- The amount of memory occupied by the integer types is system dependent.
- On a 32-bit system such as Windows, Linux, an int occupies 4 bytes (which is 32 bits) of memory.
  - Hold numbers in the range from –2,147,483,648 to 2,147,483,647.
- It occupied only 2 bytes in MS-DOS and earlier versions of OS.
- An integer constant consists of numerical digits.
- No decimal point in an integer constant, and it must lie within the range of integers.

Other Integer Types

- long and short.
- Types long and short have fixed sizes no matter what system is used.
- Type long occupies four bytes
- Type short occupies two bytes
- To create a constant of type long, use the letter L following the numerical value,
  - longvar = 7678L;

Other Integer types

- Many compilers offer integer types that explicitly specify the number of bits used.
- These type names are preceded by two underscores.
  - __int8, __int16, __int32, and __int64.
  - __int8 type corresponds to char,
  - __int16 corresponds to short

    __int32 corresponds to both int and long.
  - __int64 type holds huge integers with up to 19 decimal digits.

Character

- Stores integers that range in value from –128 to 127.
- Occupy only 1 byte (eight bits) of memory.
- More commonly used to store ASCII characters.
- Character constants use single quotation marks around a character, like ‘a’ and ‘b’.
- When the C++ compiler encounters such a character constant, it translates it into the corresponding ASCII code.
- Standard C++ provides a larger character type called wchar_t to handle non-ASCII characters (non-english language chars).

Floating point numbers

- Floating-point variables represent real numbers
- There are three kinds of floating-point variables in C++:
  - type float, type double, and type long double.
- Type float range
  - About 3.4×10^{-38} to 3.4×10^{38}, with a precision of seven digits.
  - Occupies 4 bytes (32 bits) in memory.
- double and long double require more memory space and provide a wider range of values and more precision.
- Type double
  - 8 bytes of storage
  - Range from 1.7×10^{-308} to 1.7×10^{308} with a precision of 15 digits.
- Type long double is compiler-dependent but is often the same as double.

Variable Declaration

- In C++, variable declarations are not restricted to the beginnings of blocks (before any code)
  - you may interleave declarations/statements as needed
  - it is still good style to have declarations first
- Example

```
int main() {
  int i = 5;
  cout << “Please enter the value of j”;
  int j; // Not declared at the start
  cin >> j;
  ...
```

Counter variable in For loop

- You can declare the variable(s) used in a for loop in the initialization section of the for loop
- Good when counter used in for loop only exists in for loop (variable is throw-away)
- Example

```
int main() {
        for (int i = 0; i < 5; i++)
                printf("%d\n",i);
        printf("%d\n",i); // error!!!  I belongs to scope of for()
        ...
}
```

- Variable exists only during for loop (goes away when loop ends)

Initializing global variables

- Not restricted to use constant literal values in initializing global variables, can use any evaluable expression
- Example:

```
int rows = 5;
int cols = 6;
int size = rows * cols;
void main() {
    ...
```

Dynamic Initialization

- In C variables must be initialized using constant expressions
- The C compiler fix the initialization code during compile time
- In C++, a variable can eb initialized at run time using an expression at the place of declaration.
- Dynamic initialization is extensively used in OOP.
- Example:

```
....
....
int rad;
...
...
float area = 3.04 * rad *rad;
```

void*

- Two common use of void
- Specifying return type when function returns nothing
- Declaration as generic pointer
- In C it is legal to cast other pointers to and from a void *
- In C++ this is an error, to refer the value of a variable via void pointer.
- Example:

```
int a,*ptr=NULL;
void *vptr;
a=10;
ptr=&a;
vptr= (void *)ptr;
cout << vptr << endl;
cout << *vptr;
```

NULL in C++

- C++ does not use the value NULL, instead NULL is always 0 in C++, so we simply use 0
- Example:

```
int *P = 0; // equivalent to
            // setting P to NULL
```

- Can check for a 0 pointer as if true/false:

```
if (!P)  // P is 0 (NULL)
  ...
else // P is not 0 (non-NULL)
  ...
```

enum in C++

- enum keyword in C automatically enumerates a list of word by assigning them values 0, 1, 2, …
- Enumerated types not directly represented as integers in C++
- Certain operations that are legal in C do not work in C++
- Example:
- void main() {
  - enum Color { red, blue, green };

    Color c = red;

    c = blue;

    c = 1; // Error in C++

    ++c; // Error in C++

bool

- C has no explicit type for true/false values
- C++ introduces type bool (in later versions)
  - also adds two new bool literal constants true (1) and false (0)
- Other integral types (int, char, etc.) are implicitly converted to bool when appropriate
  - non-zero values are converted to true
  - zero values are converted to false

bool operations

- Operators requiring bool value(s) and producing a bool value:
  - && (And), || (Or), ! (Not)
- Relational operators (==, !=, <, >, <=, >=) produce bool values
- Some statements expect expressions that produce bool values:

```
if (boolean_expression)
while (boolean_expression)
do … while (boolean_expression)
for ( ; boolean_expression; )
```

Operators in C++

- All C operators are valid in C++
- In addition, C++ has some new operators
  - << - Insertion operator
  - >> - Extraction operator
  - **::** - Scope resolution operator
  - **::*** - Pointer to member declarator
  - **->*** - Pointer to member operator
  - **.*** - Pointer to member operator
  - **delete** – Memory release operator
  - **new** – Memory allocation operator

Input/Output using C++

- C++ **iostream** instead of stdio.h
- Why change?
  - Input/output routines in iostream can be extended to new types declared by the user
  - The routines are in some senses easier to use

Using iostream.h

- Include iostream instead of stdio.h
- Standard iostream objects:
  - cout - object providing a connection to the monitor
  - cin - object providing a connection to the keyboard
  - cerr - object providing a connection to error stream
- To perform input and output we send messages to one of these objects (or one that is connected to a file)

Insertion operator (<<)

- To send output to the screen we use the insertion operator on the object cout
- Format: cout << *Expression*;
- The compiler figures out the type of the object and prints it out appropriately
  - cout << 5; // Outputs 5
  - cout << 4.1; // Outputs 4.1
  - cout << “String”; // Outputs String
  - cout << ‘\n’; // Outputs a newline

Extraction operator (>>)

- To get input from the keyboard we use the extraction operator and the object cin
- Format: cin >> *Variable*;
- No need for & in front of variable
- The compiler figures out the type of the variable and reads in the appropriate type
  - int X;
  - float Y;
  - cin >> X; // Reads in an integer
  - cin >> Y; // Reads in a float

Chaining calls

- Multiple uses of the insertion and extraction operator can be *chained* together:
  - cout << *E1* << *E2* << *E3* << … ;
  - cin >> *V1* >> *V2* >> *V3* >> …;
- Equivalent to performing the set of insertion or extraction operators one at a time
- Example
  - cout << “Total sales are $” << sales << ‘\n’;
  - cin >> Sales1 >> Sales2 >> Sales3;

C++ manipulators

- Manipulators are the operators used with the insertion operator (<<) to modify or manipulate the way data is displayed
- Require the header file **iomanip**
- Example
  - endl
  - setw
  - precision
  - fill

C++ manipulators - endl

- Use standard manipulator **endl** for new line feed instead of '\n'.
- Not only inserts a new-line character, but also flushes the stream.
- Example
  - cout << " Sum of the nos = " << sum << endl;
  - cout << " Largest number = " << max << endl;

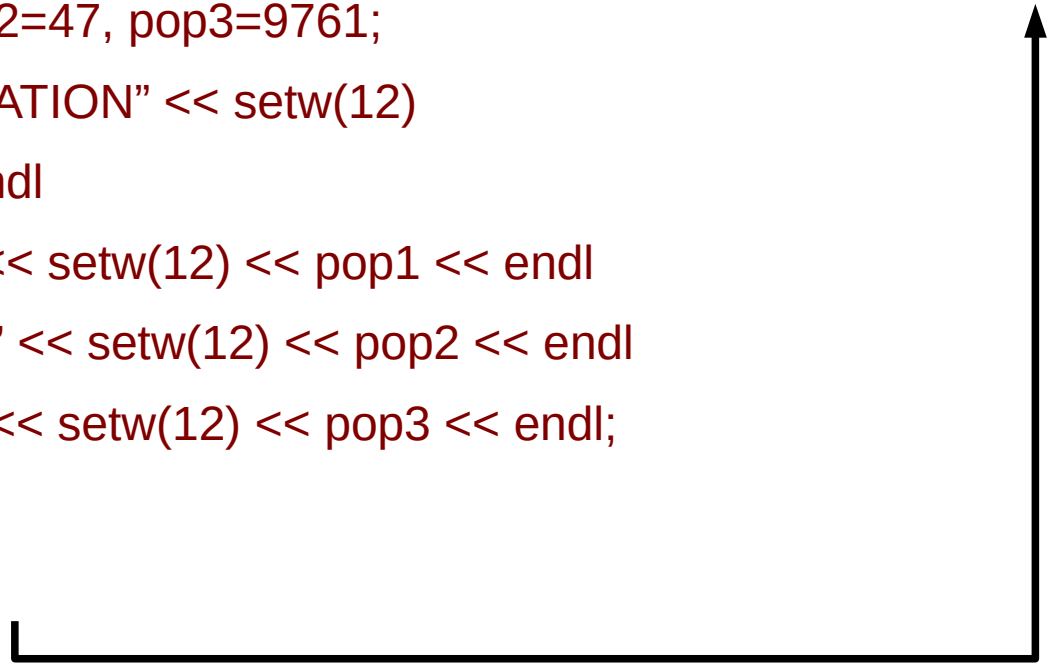
C++ manipulators - setw

- **setw** is used to define the field width of an item for the output.
- Syntax
  - cout.width(w),
  - cout << setw(*w*) << ..., set the field width of length, *w* for the output of an item.
- Specifies the width for only one item
- Revert back to default format once printed an item

setw example

```
// demonstrates setw manipulator
#include <iostream>
#include <iomanip> // for setw
using namespace std;
int main()
{
      long pop1=2425785, pop2=47, pop3=9761;
      cout << setw(8) << “LOCATION” << setw(12)
      << “POPULATION” << endl
      << setw(8) << “Portcity” << setw(12) << pop1 << endl
      << setw(8) << “Hightown” << setw(12) << pop2 << endl
      << setw(8) << “Lowville” << setw(12) << pop3 << endl;
      return 0;
}
```

```
LOCATION  POPULATION
Portcity     2425785
Hightown          47
Lowville        9761
```



C++ manipulators - precision

- precision operator is used to control the precision of floating point numbers displayed in the output
- Used to specify the number of digit to be displayed after decimal digit.
- Syntax
  - cout.precision(n);
  - cout << setprecision(n) << ...

precision example

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

int main() {
	float y = 23.1415;
	cout.precision(1);
	cout << y << endl; // Outputs 2e+01
	cout.precision(2);
	cout << y << endl; // Outputs 23
	cout.precision(3);
	cout << y << endl; // Outputs 23.1
	double f =3.14159;
	cout << setprecision(5) << f << endl;
	cout << setprecision(9) << f << endl;
	cout << fixed;
	cout << setprecision(5) << f << endl;
	cout << setprecision(9) << f << endl;
	return 0;
}
```

```
2e+01
23
23.1
3.1416
3.14159
3.14159
3.141590000
```

C++ manipulators - fill

- While displaying an item, the unsed display positions are filled with whitespace by default.
- fill manipulators allow user to fill the unsed position with desired characters.

- Syntax
  - cout.fill(ch),
  - cout << setfill << ... , fills used position by charater value of ch.

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

int main () {
  cout << setfill ('x') << setw (10);
  cout << 77 << endl;
  return 0;
}
```

Scope

- The scope of the variable extends from the point of its declaration till the end of the block containing declaration
- A variable declared inside a block is local to that block

- ...
- {
- int x =10;
- ....
- {
- ....
- int x = 1;

namespaces

- Sometimes a variable of one scope will “overlap” (i.e., collide) with a variable of the same name in a different scope.
- Overlapping can occur at many levels.
- Identifier overlapping occurs frequently in libraries using the same names for global identifiers (such as functions).
- This can cause compiler errors.
- The C++ standard solves this problem with namespaces.
- Each namespace defines a scope in which identifiers and variables are placed.

namespaces and scope resolution operator

- To use a namespace member,
  - either the member's name must be qualified with the namespace name and the **scope resolution operator** (::)
    - Example- **MyNameSpace :: member**
  - or a **using** directive must appear before the name is used in the program
    - **using namespace MyNameSpace;**

namespaces

- A using directive of the form **using std::cout**; brings one name into the scope where the directive appears.
- A using directive of the form **using namespace std;** brings all the names from the specified namespace (std) into the scope where the directive appears.

Defining namespaces

- The keyword namespace is used to define namespace.
- The body of a namespace is delimited by braces ({}).
- Syntax:
- namespace identifier {

```
        // declare entities here
}
```

- Example:

```
namespace simpleNSpace  {
      int a, b;
}
```

Defining namespaces(2)

- A namespace can be nested within another namespace
- A namespace can contain constants, data, classes, nested namespaces, functions, etc.
- Definitions of namespaces must occupy the global scope or be nested within other namespaces.

Defining namespaces(3)

- An unnamed namespace containing the members are accessible only in the current translation unit (a .cpp file and the files it includes).
- However, unlike variables, classes or functions with static linkage, those in the unnamed namespace may be used as template arguments.
- The unnamed namespace appear to occupy the global namespace, is accessible directly and does not have to be qualified with a namespace name.
- Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file.

A simple example

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

namespace first {
    int var = 5;
}
namespace second {
    double var = 3.1416;
}
int main () {
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
```

Another example

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

namespace first {
  int x = 5;
  int y = 10;
}

namespace second {
  double x = 3.1416;
  double y = 2.7183;
}
```

```
int main () {
//using namespace first;
//using namespace second;
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

A Complete Example

```
// Demonstrating namespaces.
#include <iostream>
using namespace std;

int integer1 = 98; // global variable

// create namespace Example
namespace Example {
    // declare two constants and one variable
    const double PI = 3.14159;
    const double E = 2.71828;
    int integer1 = 8;
    void printValues(); // prototype

    // nested namespace
    namespace Inner {
            enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
    } // end Inner namespace
} // end Example namespace
```

Example contd..

```
// create unnamed namespace
namespace {
    double doubleInUnnamed = 88.22; // declare variable
} // end unnamed namespace


void Example::printValues() {
     cout << "\nIn printValues:\n integer1 = " << integer1;
     cout << "\nPI = " << PI << "\nE = " << E;
     cout << "\ndoubleInUnnamed = " << doubleInUnnamed;
     cout << "\n(global) integer1 = " << ::integer1;
     cout << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
} // end printValues
```

Example contd..(2)

```
int main() {
    // output value doubleInUnnamed of unnamed namespace
    cout << "doubleInUnnamed = " << doubleInUnnamed;
    // output global variable
    cout << "\n(global) integer1 = " << integer1;

    // output values of Example namespace
    cout << "\nPI = " << Example::PI << "\nE = " << Example::E << endl;
    cout << "\ninteger1 = " << Example::integer1 << endl;
    cout << "\nFISCAL3 = " << Example::Inner::FISCAL3 << endl;

    Example::printValues(); // invoke printValues function

} // end main
```

new and delete operator

- Though C++ support **malloc()**, **calloc()**, and **free()** function, it defined two unary operators
  - new
  - delete
- An object can be created using new
- Destroyed by using delete, as and when required
- An object created inside a block with new, will remain inexistence until it is explicitly destroyed by using delete.

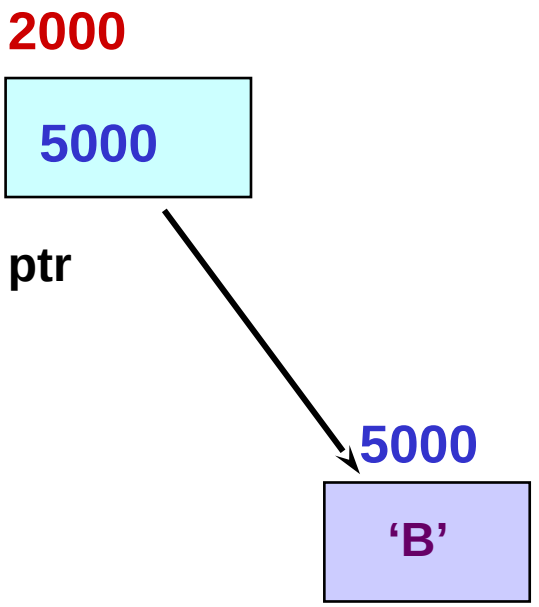
new operator syntax

- **new DataType**
- **new DataType [int / expression]**
- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- Otherwise, program terminates with error message

new operator

```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
```



new operator(2)

- Alternatively,
  - int *ptr1 = new int;
  - float *ptr2 = new float;
- Allocation cum initialization
  - ptr-variable = new data-type(value);
  - int *ptr1 = new int(25);
  - float *ptr2 = new float(7.5);

- Allocating one dimensional array
  - ptr-variable = new data-type[size];
  - int *arr_ptr1 = new int[10];
  - float* farray = new float[size*2];

Allocating 2D array using new

- Via pointer to an array
  - int (*ptr)[5] = new int[4][5];
- Via array

Type ***twodname*;

twodname = new *Type**[*dimension1*];

for (int d1 = 0; d1 < *dimension1*; d1++)

twodname[d1] = new *Type*[*dimension2*];

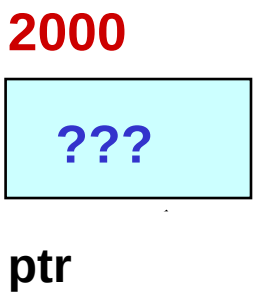
delete operator syntax

- **delete Pointer**
- **delete [ ] Pointer**
- The object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.
- Good idea to set the pointer to the released memory to NULL
- Square brackets are used with delete to deallocate a dynamically allocated array.

delete operator

```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
cout  <<  *ptr;
delete  ptr;
```



delete for 2D array

```
for (int d1 = 0; d1 < dimension1; d1++)
    delete [] twodname[d1];
delete [] twodname;
```

2D array example

```
 int **A;

A = new int*[5];
for (int i = 0; i < 5; i++) {
        A[i] = new int[8];
        for (int j = 0; j < 8; j++)
            A[i][j] = i + j;
}

for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 8; j++)
            cout << setw(3) << A[i][j];
        cout << endl;
}

for (int i = 0; i < 5; i++)
        delete [] A[i];
delete [] A;
```

Reference variable

- A reference variable provides an alias (alternative name) for a previously defined variable.
  - datatype &reference_name = variable_name
- Example:
  - float total = 100;
  - float &sum = total;
- Both variables refers to the same data object in the memory.

Reference variable

- Any change by either of the variables will reflect the change on the data object in the memory.

```
int x = 5;
```

- `int &z = x;` // z is another name for x
- `int &y ;` //Error: reference must be initialized
- `cout << x << endl;` // prints 5
- `cout << z << endl;` // prints 5
-
- `z = 9;` // same as x = 9;
-
- `cout << x << endl;` // prints 9
- `cout << z << endl;` // prints **9**

Reference variable

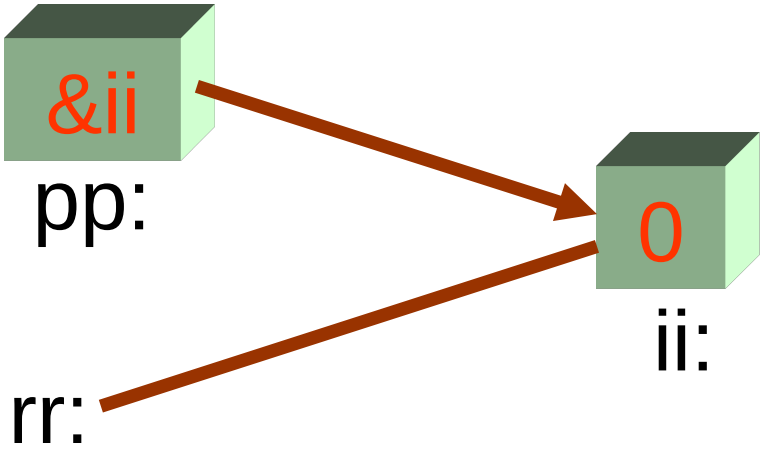
- To ensure that a reference is a name for something(bound to an object), we must initialize the reference
- Example:
  - int i=1;
  - int& r2; //wrong
  - Int& r1=i; //OK

Why reference variable?

- Are primarily used as function parameters
- Advantages of using references:
  - you don’t have to pass the address of a variable
  - you don’t have to dereference the variable inside the called function

Reference and Pointers

```
int ii = 0;
int& rr=ii;
rr++;
int *pp=&rr // or &ii
```



- pp is a variable which stores address of another variable
- rr is an alternative name (alias) for an existing variable
- The value of a reference cant be changed after initialization. It always refers to the same object it was initialized. Which is not the case in pointers.

swap example with reference

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main()
{
	int x = 5, y = 10;
	cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
	swap(x,y);
	cout << "Main. After swap, x: " << x << " y: " << y << "\n";
	return 0;
}
void swap (int &rx, int &ry)
{
	int temp;
	cout << "Swap. Before swap, rx: " << rx << " ry: " << ry << "\n";
	temp = rx;
	rx = ry;
	ry = temp;
	cout << "Swap. After swap, rx: " << rx << " ry: " << ry << "\n";
}
```

Initializing Array elements

- When giving a list of initial array values in C++, you can use expressions that have to be evaluated
- Values calculated at run-time before initialization done
- Example:

```
int main() {
  int n1, n2, n3;
  int *nptr[] = { &n1, &n2, &n3 };
   ...
```

Use of const

- Only one method in C:
  - #define ArraySize 100; //Macro constants
- Another way in C++:
  - const ArraySize =100;
- Constant can be used in local scope and is often used when the value cannot be changed
- In C++, const can also be used in a function prototype to prevent argument to from being modified within the function.
- C++ const are local to the file in which they are declared.
- Example
  - const float PI = 3.14159F;

const vs. #define

- Unlike const, you can’t specify the data type of the constant using #define
- It may can lead to program bugs;
- This is why #define has been superseded by const used with normal variables.

Revision of structure

- A structure is a derived data type used to handled a collection of logically related data.
- The variables in a structure can be of different types:
- The data items in a structure are called the members of the structure.
- A structure (as typically used) is a collection of data, while a class is a collection of both data and functions.
- A structure is defined using **struct** keyword
- Once the structure type has been defined, we can create variables of structure type using similar declaration as built-in type.

Simple example

```
// uses parts inventory to demonstrate
structures
#include <iostream>
using namespace std;
struct part //declare a structure
{
    int modelnumber; //ID number of widget
    int partnumber; //ID number of widget part
    float cost; //cost of part
};

int main(){
    part part1; //define a structure variable
    part1.modelnumber = 6244;
    part1.partnumber = 373;
    part1.cost = 217.55F;

    //display structure members
    cout << “Model “ << part1.modelnumber;
    cout << “, part “ << part1.partnumber;
    cout << “, costs $” << part1.cost << endl;
    return 0;
}
```

structure declaration

structure variable definition

Any difference did you find in this program compared to C ???

Example contd..

- part part1;
- int var1;
- This similarity is not accidental.
- One of the aims of C++ is to make the syntax and the operation of user-defined data types as similar as possible to that of built-in data types.
- In C we need to include the keyword struct in structure definitions,
  - struct part part1;
- In C++ the keyword is not necessary.

Accessing Structure Members

- Once a structure variable has been defined, its members can be accessed using called the dot operator
  - part1.modelnumber = 6244;
- The structure member is written in three parts:
  - The name of the structure variable (part1);
  - The dot operator, which consists of a period (.); and
  - The member name (modelnumber).
- A pointer to a structure variable uses -> operator to access the members
  - part *ptr_part;
  - ptr_part->modelnumber = 6244;

Initializing Structure Members

```
#include <iostream>
using namespace std;
struct part {
    int modelnumber; //ID number of widget
    int partnumber; //ID number of widget
part
    float cost; //cost of part
};

int main() {
    //initialize variable
    part part1 = { 6244, 373, 217.55F };
    part part2;
    //display first variable
    cout << “Model “ << part1.modelnumber;
    cout << “, part “ << part1.partnumber;
    cout << “, costs $” << part1.cost << endl;
    part2 = part1;
    //assign first variable to second
    //display second variable
    cout << “Model “ << part2.modelnumber;
    cout << “, part “ << part2.partnumber;
    cout << “, costs $” << part2.cost << endl;
    return 0;
}
```

Structures Within Structures

```
// demonstrates nested structures
#include <iostream>
using namespace std;

struct Distance {
    int feet;
    float inches;
};

struct Room {
    Distance length;
    Distance width;
};
```

```
int main() {
    Room dining; //define a room
    dining.length.feet = 13; //assign values to room
    dining.length.inches = 6.5;
    dining.width.feet = 10;
    dining.width.inches = 0.0;

    //convert length and width
    float l = dining.length.feet + dining.length.inches/12;
    float w = dining.width.feet + dining.width.inches/12;

    cout << “Dining room area is “ << l * w << “ square feet\n” ;
    return 0;
}
```