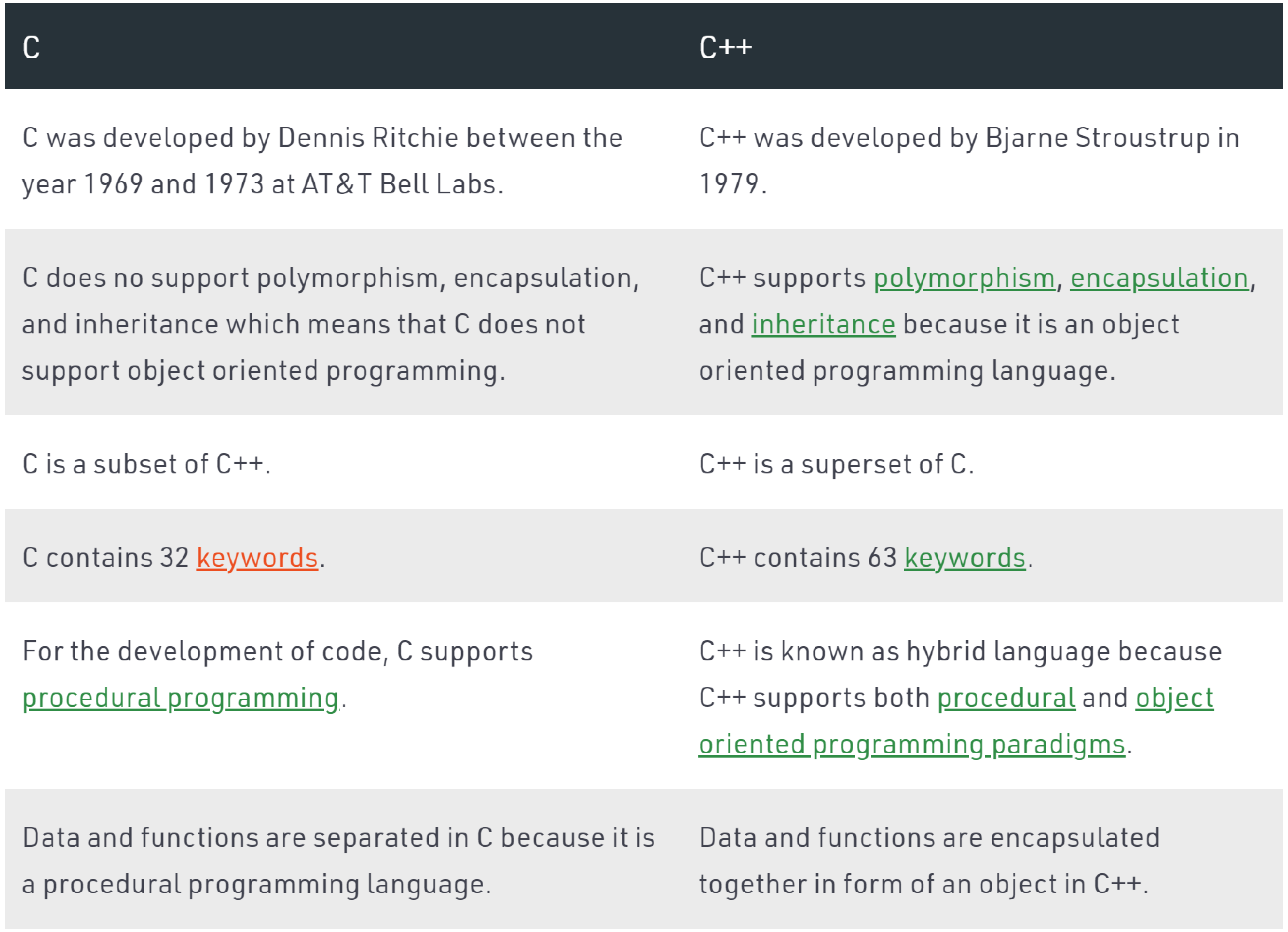
THE BIOROBOTICS INSTITUTE

Sant'Anna School of Advanced Studies – Pisa

Introduction to C++ concepts

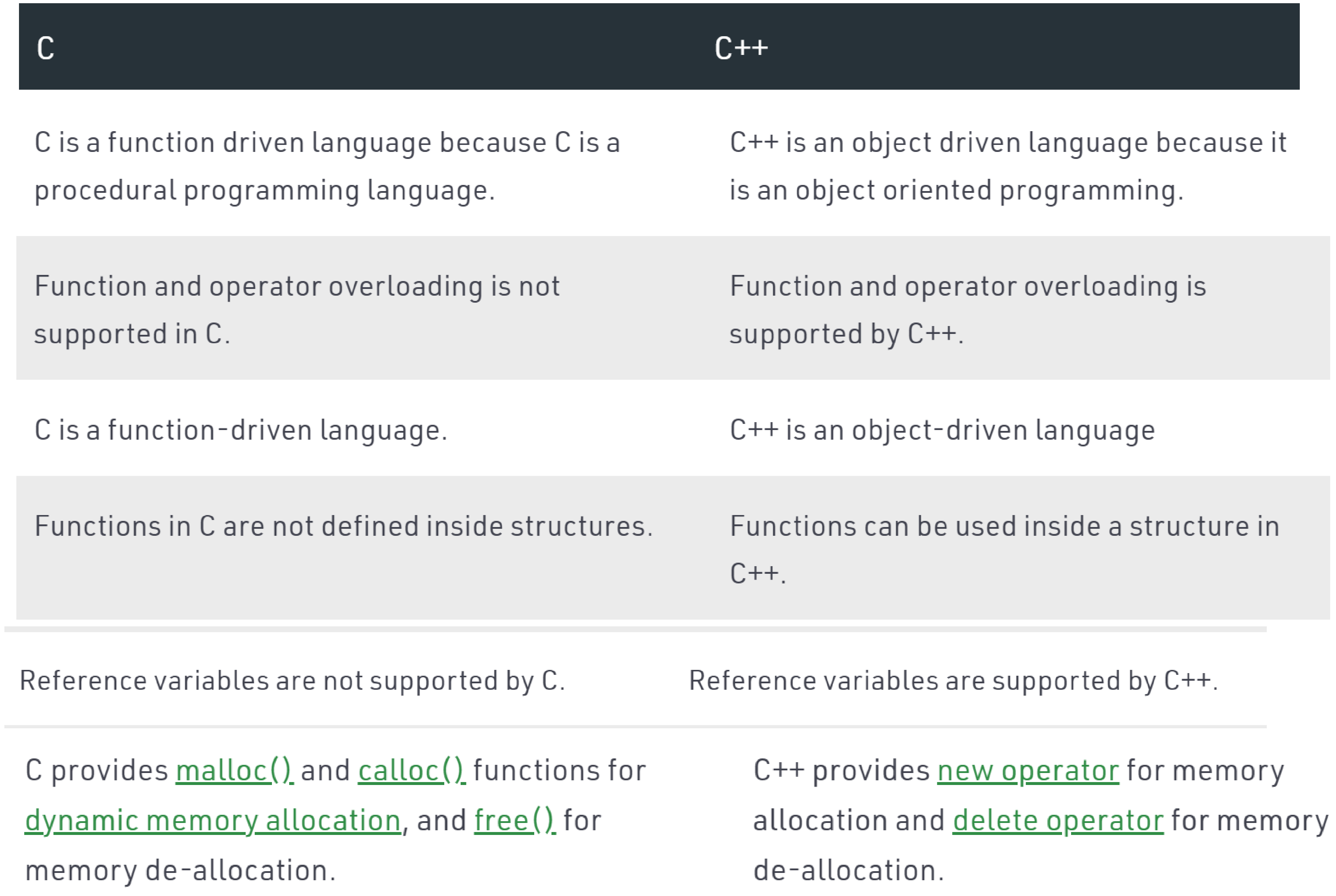
Egidio Falotico

C vs C++



C	C++
C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.
C does no support polymorphism, encapsulation, and inheritance which means that C does not support object oriented programming.	C++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language.
C is a subset of C++.	C++ is a superset of C.
C contains 32 keywords.	C++ contains 63 keywords.
For the development of code, C supports procedural programming.	C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.
Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in form of an object in C++.

C vs C++



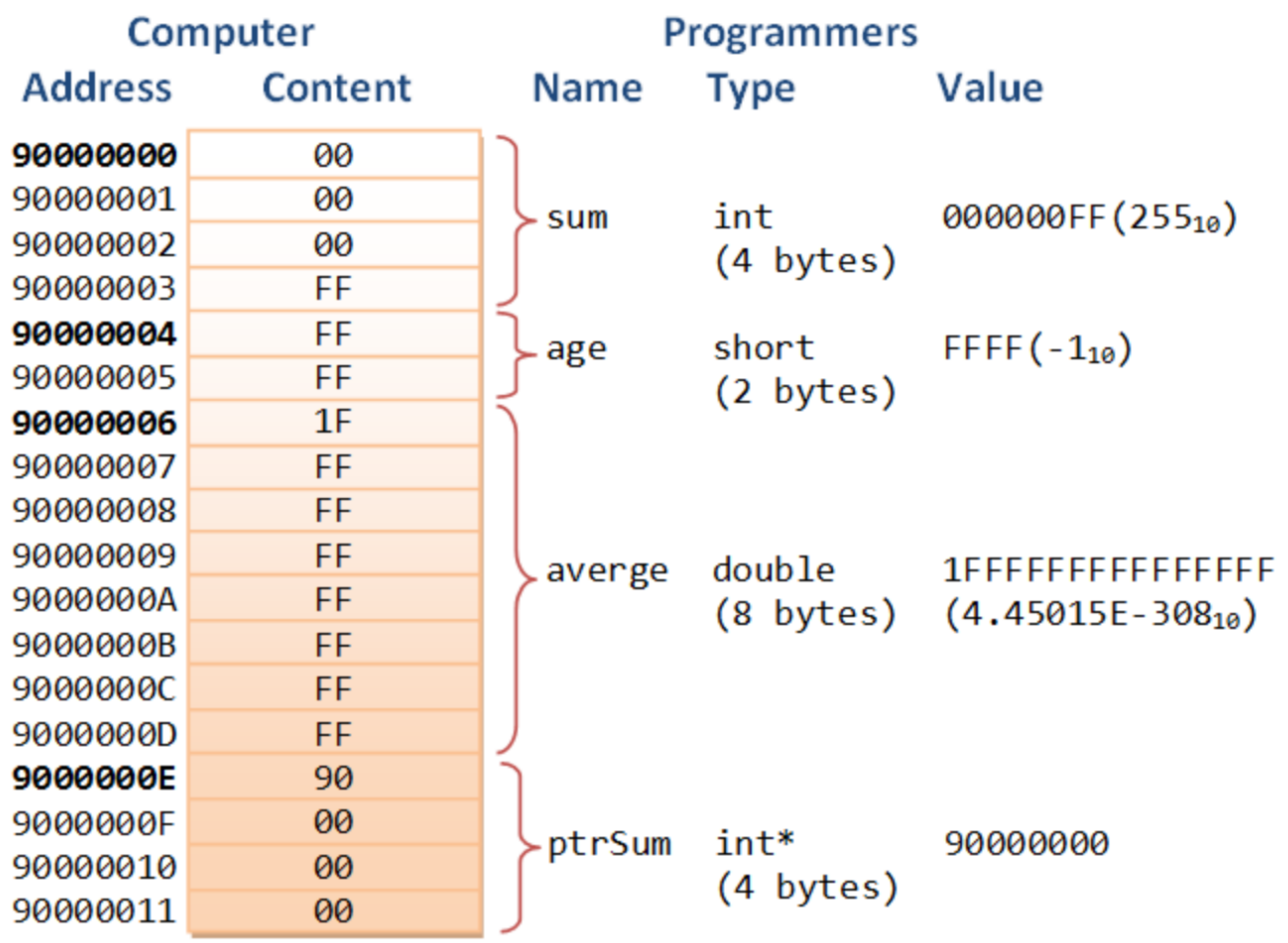
C	C++
C is a function driven language because C is a procedural programming language.	C++ is an object driven language because it is an object oriented programming.
Function and operator overloading is not supported in C.	Function and operator overloading is supported by C++.
C is a function-driven language.	C++ is an object-driven language
Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Reference variables are not supported by C.	Reference variables are supported by C++.
C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.

Pointer Variables

- A computer memory location has an address and holds a content. The address is a **numerical number** (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data.

- To ease the burden of programming using numerical address and programmer-interpreted data, early programming languages (such as C) introduce the concept of variables. A variable is a named location that can store a value of a particular type. Instead of numerical addresses, names (or identifiers) are attached to certain addresses. Also, types (such as int, double, char) are associated with the contents for ease of interpretation of data.

Pointer Variables



Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF(255$_{10}$)
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1$_{10}$)
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF (4.45015E-308$_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Pointer Declaration

Pointers must be declared before they can be used, just like a normal variable. The syntax of declaring a pointer is to place a * in front of the name. A pointer is associated with a type (such as int and double) too.

```
type *ptr;
// or
type* ptr;
// or
type * ptr;
```

```
int *p1, *p2, i;    // p1 and p2 are int pointers. i is an int
int* p1, p2, i;     // p1 is a int pointer, p2 and i are int
int * p1, * p2, i;  // p1 and p2 are int pointers, i is an int
```

Initializing Pointers

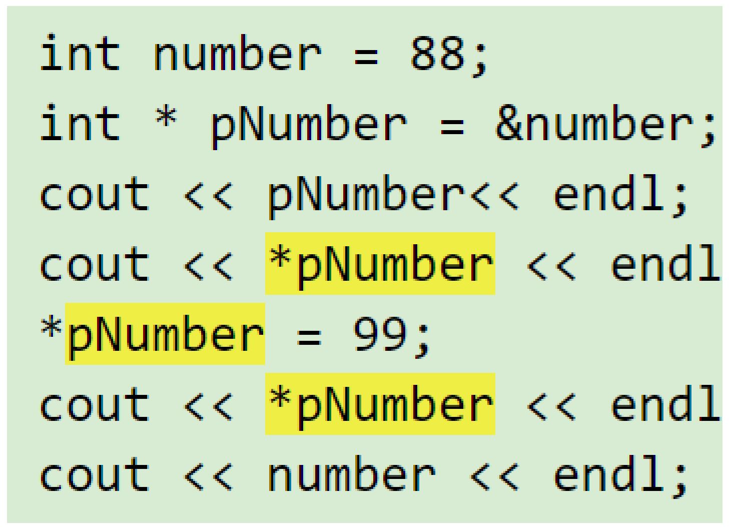
When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location.
This is normally done via the address-of operator (&).

The address-of operator (**&**) operates on a variable, and returns the address of the variable. For example, if number is an int variable, **&number** returns the address of the variable number.

```
int number = 88;
int * pNumber;
pNumber = &number;
```

Indirection or deferencing pointer

The indirection operator (or dereferencing operator) (*) operates on a pointer, and returns the value stored in the address kept in the pointer variable. For example, if *pNumber* is an *int* pointer, **pNumber* returns the int value "pointed to" by *pNumber*.



```
int number = 88;
int * pNumber = &number;
cout << pNumber<< endl;
cout << *pNumber << endl
*pNumber = 99;
cout << *pNumber << endl
cout << number << endl;
```

The symbol * has different meaning in a declaration statement and in an expression. When it is used in a declaration (e.g., int * pNumber), it denotes that the name followed is a pointer variable.

Whereas when it is used in a expression (e.g., *pNumber = 99; temp << *pNumber;), it refers to the value pointed to by the pointer variable.

Pointer Type

```
int i = 88;
double d = 55.66;
int * iPtr = &i;    // int pointer pointing to an int value
double * dPtr = &d; // double pointer pointing to a double value

iPtr = &d;   // ERROR, cannot hold address of different type
dPtr = &i;   // ERROR
iPtr = i;    // ERROR, pointer holds address of an int, NOT int value

int j = 99;
iPtr = &j;  // You can change the address stored in a pointer
```

Pointer - Example

```cpp
/* Test pointer declaration and initialization (TestPointerInit.cpp) */
#include <iostream>
using namespace std;

int main() {
   int number = 88;    // Declare an int variable and assign an initial value
   int * pNumber;      // Declare a pointer variable pointing to an int (or int pointer)
   pNumber = &number;  // assign the address of the variable number to pointer pNumber

   cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)
   cout << &number << endl;  // Print address of number (0x22ccf0)
   cout << *pNumber << endl; // Print value pointed to by pNumber (88)
   cout << number << endl;   // Print value of number (88)

   *pNumber = 99;            // Re-assign value pointed to by pNumber
   cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)
   cout << &number << endl;  // Print address of number (0x22ccf0)
   cout << *pNumber << endl; // Print value pointed to by pNumber (99)
   cout << number << endl;   // Print value of number (99)
                             // The value of number changes via pointer

   cout << &pNumber << endl; // Print the address of pointer variable pNumber (0x22ccec)
}
```

Reference Variables

- C++ added the so-called *reference variables* (or *references* in short). A reference is an *alias*, or an *alternate name* to an existing variable.

- The meaning of symbol & is different in an expression and in a declaration. When it is used in an expression, & denotes the address-of operator, which returns the address of a variable, e.g., if number is an int variable, &number returns the address of the variable number (this has been described in the above section).

- However, when & is used in a declaration (including function formal parameters), it is part of the type identifier and is used to declare a reference variable (or reference or alias or alternate name). It is used to provide another name, or another reference, or alias to an existing variable.

References and Pointers

```cpp
/* References vs. Pointers (TestReferenceVsPointer.cpp) */
#include <iostream>
using namespace std;

int main() {
   int number1 = 88, number2 = 22;

   // Create a pointer pointing to number1
   int * pNumber1 = &number1;  // Explicit referencing
   *pNumber1 = 99;             // Explicit dereferencing
   cout << *pNumber1 << endl;  // 99
   cout << &number1 << endl;   // 0x22ff18
   cout << pNumber1 << endl;   // 0x22ff18 (content of the pointer variable - same as above)
   cout << &pNumber1 << endl;  // 0x22ff10 (address of the pointer variable)
   pNumber1 = &number2;        // Pointer can be reassigned to store another address

   // Create a reference (alias) to number1
   int & refNumber1 = number1;  // Implicit referencing (NOT &number1)
   refNumber1 = 11;             // Implicit dereferencing (NOT *refNumber1)
   cout << refNumber1 << endl;  // 11
   cout << &number1 << endl;    // 0x22ff18
   cout << &refNumber1 << endl; // 0x22ff18
   //refNumber1 = &number2;     // Error! Reference cannot be re-assigned
                                // error: invalid conversion from 'int*' to 'int'
   refNumber1 = number2;        // refNumber1 is still an alias to number1.
                                // Assign value of number2 (22) to refNumber1 (and number1).
   number2++;
   cout << refNumber1 << endl;  // 22
   cout << number1 << endl;     // 22
   cout << number2 << endl;     // 23
}
```

Pass by value

In C/C++, by default, arguments are passed into functions *by value* (except arrays which is treated as pointers). That is, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function has no effect to the original argument in the caller. In other words, the called function has no access to the variables in the caller.

Pass by value

```
/* Pass-by-value into function (TestPassByValue.cpp) */
#include <iostream>
using namespace std;

int square(int);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;         // 8
   cout << square(number) << endl; // 64
   cout << number << endl;         // 8 - no change
}

int square(int n) {  // non-const
   cout <<  "In square(): " << &n << endl;  // 0x22ff00
   n *= n;           // clone modified inside the function
   return n;
}
```

Pass by Reference with pointers

In many situations, we may wish to **modify the original copy** directly (especially in passing huge object or array) to avoid the overhead of cloning. This can be done by passing a **pointer** of the object into the function, known as *pass-by-reference*.

Pass by Reference with Pointers

```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */
#include <iostream>
using namespace std;

void square(int *);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;   // 8
   square(&number);          // Explicit referencing to pass an address
   cout << number << endl;   // 64
}

void square(int * pNumber) {  // Function takes an int pointer (non-const)
   cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
   *pNumber *= *pNumber;      // Explicit de-referencing to get the value pointed-to
}
```

Pass-by-Reference with Reference

Instead of passing pointers into function, you could also pass references into function, to avoid the clumsy syntax of referencing and dereferencing.

Pass-by-Reference with Reference

```cpp
/* Pass-by-reference using reference (TestPassByReference.cpp) */
#include <iostream>
using namespace std;

void square(int &);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;  // 8
   square(number);          // Implicit referencing (without '&')
   cout << number << endl;  // 64
}

void square(int & rNumber) {  // Function takes an int reference (non-const)
   cout <<  "In square(): " << &rNumber << endl;  // 0x22ff1c
   rNumber *= rNumber;        // Implicit de-referencing (without '*')
}
```

Difference in reference variables and pointer variable

- References are generally implemented using pointers. A reference is same object, just with a different name and reference must refer to an object. Since references can’t be NULL, they are safer to use.
- A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
- Pointer can be assigned NULL directly, whereas reference cannot.
- Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
- A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.

Array in C++

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows

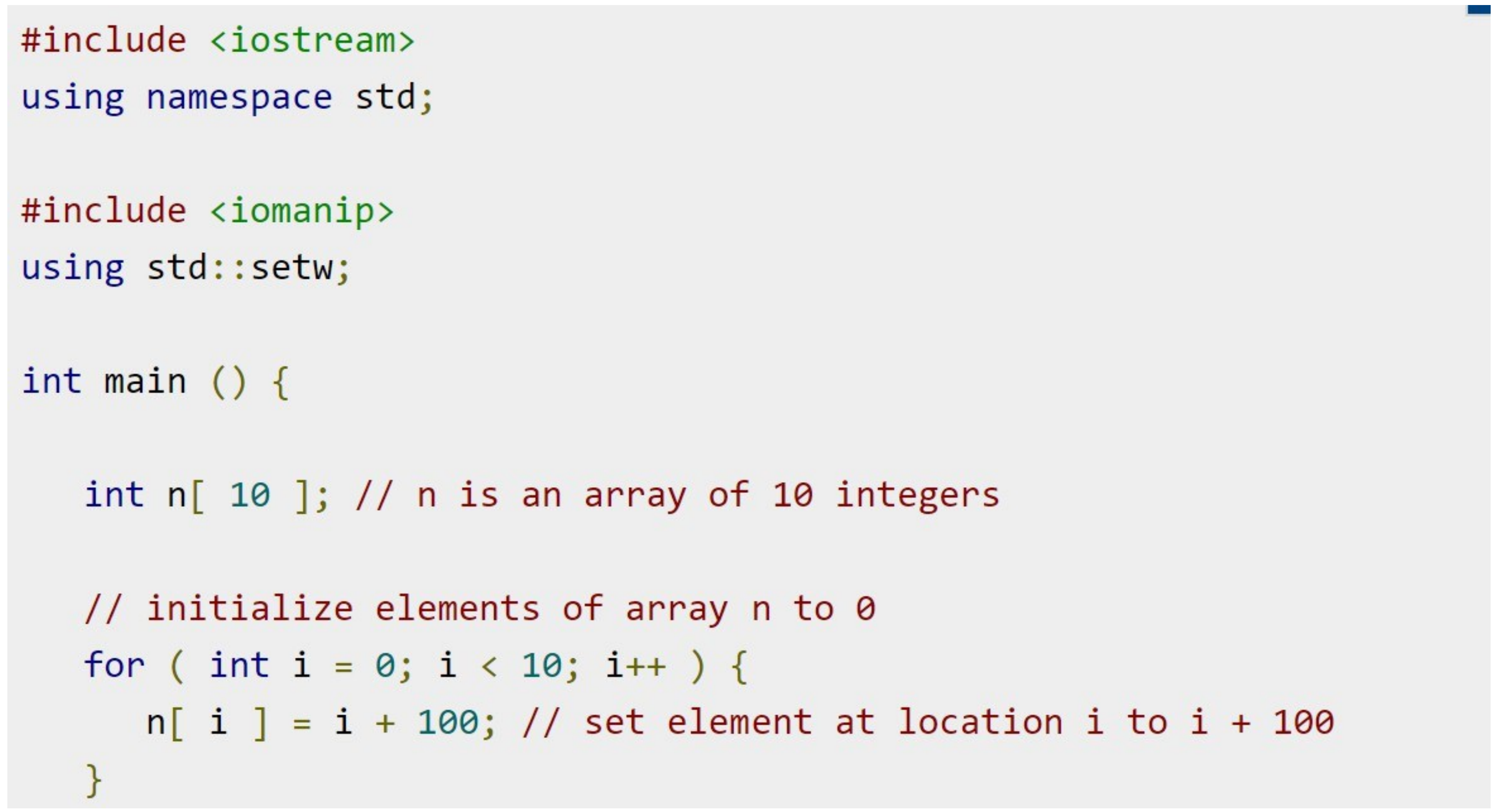
```
type arrayName [ arraySize ];
```

You can initialize C++ array elements either one by one or using a single statement as follows

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example: `double salary = balance[9];`



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

#include <iomanip>
using std::setw;

int main () {

   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // set element at location i to i + 100
   }
```

Pointers and arrays

A pointer stores the address of a single variable and it can also store the address of cells of an array.

ptr is a pointer variable while *arr* is an int array. The code *ptr=arr;* stores the **address** of the first element of the array in variable *ptr*.

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

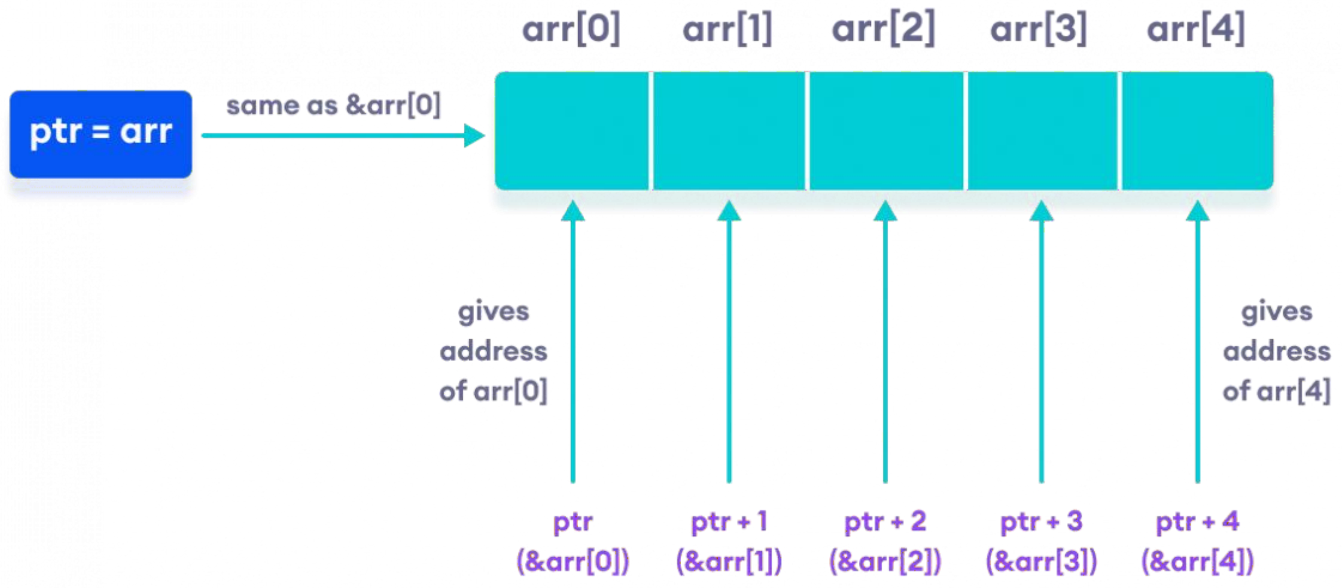
We use *arr* instead of *&arr[0]*. This is because both are **the same**.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

Point to array elements

We want a pointer to point to a specific element of the array.

Ex. If *ptr* points to the first element of the array, *ptr+2* points to the third elements



```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

```
// use dereference operator
*ptr == arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```

Basic Input/output

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file.

A *`stream`* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access sources and destinations of characters:

stream	description
`cin`	standard input stream
`cout`	standard output stream
`cerr`	standard error (output) stream
`clog`	standard logging (output) stream

Standard output (cout)

The **<<** operator inserts the data that follows it into the stream that precedes it. In the examples below, it inserted the literal string Output sentence, the number 120, and the value of variable x into the standard output stream cout. Notice that the sentence in the first statement is enclosed in double quotes (") because it is a string literal, while in the last one, x is not.

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints the value of x on screen
```

Multiple insertion operations (<<) may be chained in a single statement

```
cout << "This " << " is a " << "single C++ statement";
```

Standard output (cout)

To insert **a line break**, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as \n (i.e., a backslash character followed by a lowercase n). For example:

```
cout << "First sentence.\n";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Standard input

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is *cin*.

For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

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

Standard input/output example

```cpp
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.
```

Exercise

Make an array of 5 values, fill it in with input from keyboard and check if it is palindrome. Use pointers to access the array.

C++ class

- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.
- When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

C++ class (II)

- A class definition starts with the keyword **class** followed by the class name and the body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations.

```
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

Define Objects

- We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Class Member function

- A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

```
class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box
      double getVolume(void);// Returns box volume
};
```

Class Member Function

Member functions can be defined within the class definition or separately using **scope resolution operator**

```
class Box {
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box

      double getVolume(void) {
         return length * breadth * height;
      }
};
```

```
double Box::getVolume(void) {
   return length * breadth * height;
}
```

Example of class definition

```
class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void) {
   return length * breadth * height;
}

void Box::setLength( double len ) {
   length = len;
}
void Box::setBreadth( double bre ) {
   breadth = bre;
}
void Box::setHeight( double hei ) {
   height = hei;
}
```

```
// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
```

Class Access Modifiers

- Data hiding allows preventing the functions of a program to access directly the internal representation of a class type.
- The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.
- Default access is **private**

Class Access Modifiers

```cpp
class Line {
   public:
      double length;
      void setLength( double len );
      double getLength( void );
};

// Member functions definitions
double Line::getLength(void) {
   return length ;
}

void Line::setLength( double len) {
   length = len;
}

// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;

   return 0;
}
```

Private Members

- A **private** member variable or function cannot be accessed, or even viewed from outside the class.

```
class Box {
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );

   private:
      double width;
};

// Member functions definitions
double Box::getWidth(void) {
   return width ;
}

void Box::setWidth( double wid ) {
   width = wid;
}
```

```
// Main function for the program
int main() {
   Box box;

   // set box length without member function
   box.length = 10.0; // OK: because length is public
   cout << "Length of box : " << box.length <<endl;

   // set box width without member function
   // box.width = 10.0; // Error: because width is private
   box.setWidth(10.0);  // Use member function to set it.
   cout << "Width of box : " << box.getWidth() <<endl;

   return 0;
}
```

Protected Members

- A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

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

class Box {
   protected:
      double width;
};

class SmallBox:Box { // SmallBox is the derived class.
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void) {
   return width ;
}

void SmallBox::setSmallWidth( double wid ) {
   width = wid;
}
```

```
// Main function for the program
int main() {
   SmallBox box;

   // set box width using member function
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth() << endl;

   return 0;
}
```

Class Constructor

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
- A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.
- Constructor is usually public, but you can make it private. In such a case you cannot use it outside the class.

Constructor

```
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor
   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}
```

```
// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

Parametrized Constructor

```cpp
// Member functions definitions including constructor
Line::Line( double len) {
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
```

```cpp
Line::Line( double len): length(len) {
   cout << "Object is being created, length = " << len << endl;
}
```

```cpp
// Main function for the program
int main() {
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

Class Destructor

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Class Destructor

```
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}
```

```
// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

Use of *this*

- Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Use of *this*

```cpp
class Box {
   public:
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }
      int compare(Box box) {
         return this->Volume() > box.Volume();
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

```cpp
int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   if(Box1.compare(Box2)) {
      cout << "Box2 is smaller than Box1" <<endl;
   } else {
      cout << "Box2 is equal to or larger than Box1" <<endl;
   }

   return 0;
}
```

Pointer to a Class

- A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Pointer to a class

```cpp
int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   Box *ptrBox;                // Declare pointer to a class.

   // Save the address of first object
   ptrBox = &Box1;

   // Now try to access a member using member access operator
   cout << "Volume of Box1: " << ptrBox->Volume() << endl;

   // Save the address of second object
   ptrBox = &Box2;

   // Now try to access a member using member access operator
   cout << "Volume of Box2: " << ptrBox->Volume() << endl;

   return 0;
}
```

```cpp
class Box {
   public:
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

Static Members

- When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.
- A **static member function** can only access **static** data **member**, other **static member functions** and any other **functions** from outside the class.

Static Members

```cpp
class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;

         // Increase every time object is created
         objectCount++;
      }
      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

```cpp
// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

Static Member Functions

```
static int getCount() {
   return objectCount;
}
```

```
int main(void) {
   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

Inheritance

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Inheritance

```
// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
```

```
int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Interfaces in C++ (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration

Abstract Classes

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit.

Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

If a subclass of an Abstract Class needs to be instantiated, it has to implement **each of the virtual functions**, which means that it supports the interface

Abstract Class - Example

```cpp
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};
```

```cpp
int main(void) {
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

```
Total Rectangle area: 35
Total Triangle area: 17
```

Dynamic Memory

There may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

Dynamic memory is allocated using operator ***new***. ***new*** is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets ***[]***. It returns a pointer to the beginning of the new block of memory allocated.

Dynamic Memory syntax

Its syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where *number_of_elements* is an integer value representing the amount of these. For example:

```
int * foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to *foo* (a pointer). Therefore, *foo* now points to a valid block of memory with space for five elements of type int

Deallocate memory

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

```
delete pointer;
delete[] pointer;
```

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

Random values

srand() and *rand()* functions

The srand() function in C++ can perform pseudo-random number calculation. This function requires a seed value which forms the basis of computation of random numbers.

```
srand(unsigned int seed_value)
```

With the help of the seed value, srand() sets the stage for the generation of pseudo-random numbers by the rand() function.

```
int random = rand();
```

Generation of a random number.

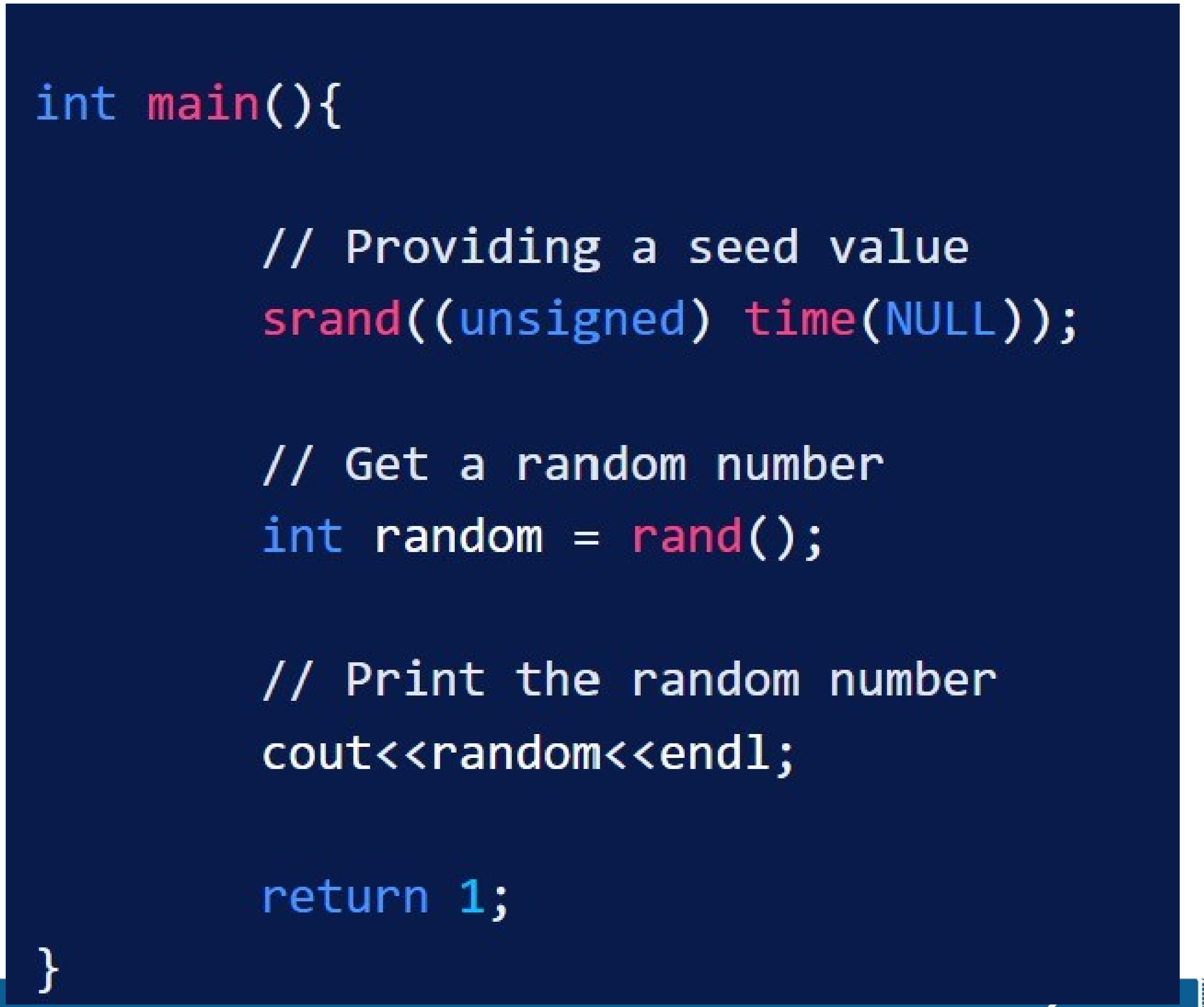
Random values

The current_time variable holds the number of seconds passed since January, 1970. This value is passed to the *srand()* function and then we get a fresh sequence of pseudo-random numbers.

We can skip the initialization of timestamp to a variable and simply pass the timestamp to the function.

srand((unsigned) time(NULL));

The seed value is provided once in a program no matter how many random numbers are to be generated.



```
int main(){

        // Providing a seed value
        srand((unsigned) time(NULL));

        // Get a random number
        int random = rand();

        // Print the random number
        cout<<random<<endl;

        return 1;
}
```

Exercise

A class (FibonacciArray) with one constructor:

-Arguments: index of Fibonacci sequence, the number to which that index corresponds is the size of the array. Indexes < 7 are discarded and the index used is 7.

Index	1	2	3	4	5	6	7	8	9	10
Array size	0	1	1	2	3	5	8	13	21	34

Half of array is filled in with the Fibonacci sequence, the other half with random numbers between 1 and 100.

Member functions:

1)static Num_obj: returns how many objects are created for that class

2)Fib_mean: computes the mean of the array

3)Fib_seventh: computes the seventh value, if they were sorted

4)Compare_dyn_mean:compare the mean of the array with the one of another object of the same class

Test it in a main function by creating 2 objects

Template (generic programming)

Generic programming is based on passing a type (such as int, double, and Point) into template. We can program in generic type and invoke the code using a specific type.

The goal of generic programming is to write code that is **independent of the data types**.

In C language, all codes are tied to a **specific** data type. For container data structures (such as array and structure), you need to specify the type of the elements.

Template (generic programming)

Template supports so-called *parameterized type* - i.e., you can use type as argument in building a class or a function (in class template or function template). Template is extremely useful if a particular algorithm is to be applied to a variety of types, e.g., a *container* class which contains elements, possibly of various types.

C++'s Standard Template Library (STL) provides template implementation of many *container* classes, such as **vector**, which can be used to hold elements of all types. STL also provides generic representation of algorithm (such as searching and sorting), which works on the generic container**.**

Vector template Class

C/C++ built-in array has many drawbacks:

1.It is fixed-size and needs to be allocated with the fixed-size during declaration. It does not support *dynamic allocation*. You cannot increase the size of an array during execution.

2.Array does not provide index-bound check. You could use an index which is outside the array's bound.

3.You need to roll your own codes to compare two arrays (via ==), copy an array into another array (via assignment =), etc.

C++ provides a vector template class, as part of Standard Template Library (STL). It is defined in header **<vector>,** belonging to the namespace std.
vector is the most commonly used STL class, which provides an alternative to array, supports dynamic memory allocation and many operations (such as comparison and assignment).
vector is a template class, which can be instantiated with a type, in the format: vector<int>, vector<double>, vector<string>. The same template class can be used to handle many types, instead of repeatably writing codes for each of the type.

```
/* Test vector template class (TestVectorTemplate.cpp) */
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void print(const vector<int> & v);

int main() {
   vector<int> v1(5);  // Create a vector with 5 elements;

   // Assign values into v1, using array-like index []
   // You can retrieve the size of vector via size()
   for (int i = 0; i < v1.size(); ++i) {
      v1[i] = (i+1) * 2;          // no index-bound check for []
   }

   // Print vector content, using at()
   for (int i = 0; i < v1.size(); ++i) {
      cout << v1.at(i) << " ";    // do index-bound check with at()
   }
   cout << endl;

   vector<int> v2;   // Create a vector with 0 elements
   // Assign v1 to v2 memberwise
   v2 = v1;
   for (int i = 0; i < v2.size(); ++i) {
      cout << v2[i] << " ";
   }
   cout << endl;

   // Compare 2 vectors memberwise
   cout << boolalpha << (v1 == v2) << endl;

   // Append more elements - dynamically allocate memory
   v1.push_back(80);
   v1.push_back(81);
   for (int i = 0; i < v1.size(); ++i) {
      cout << v1[i] << " ";
   }
   cout << endl;

   // Remove element from the end
   v1.pop_back();
   for (int i = 0; i < v1.size(); ++i) {
      cout << v1[i] << " ";
   }
   cout << endl;

   vector<string> v3;  // Create a vector of string with 0
   v3.push_back("A for Apple");    // append new elements
   v3.push_back("B for Boy");
   for (int i = 0; i < v3.size(); ++i) {
      cout << v3[i] << " ";
   }
   cout << endl;
}
```

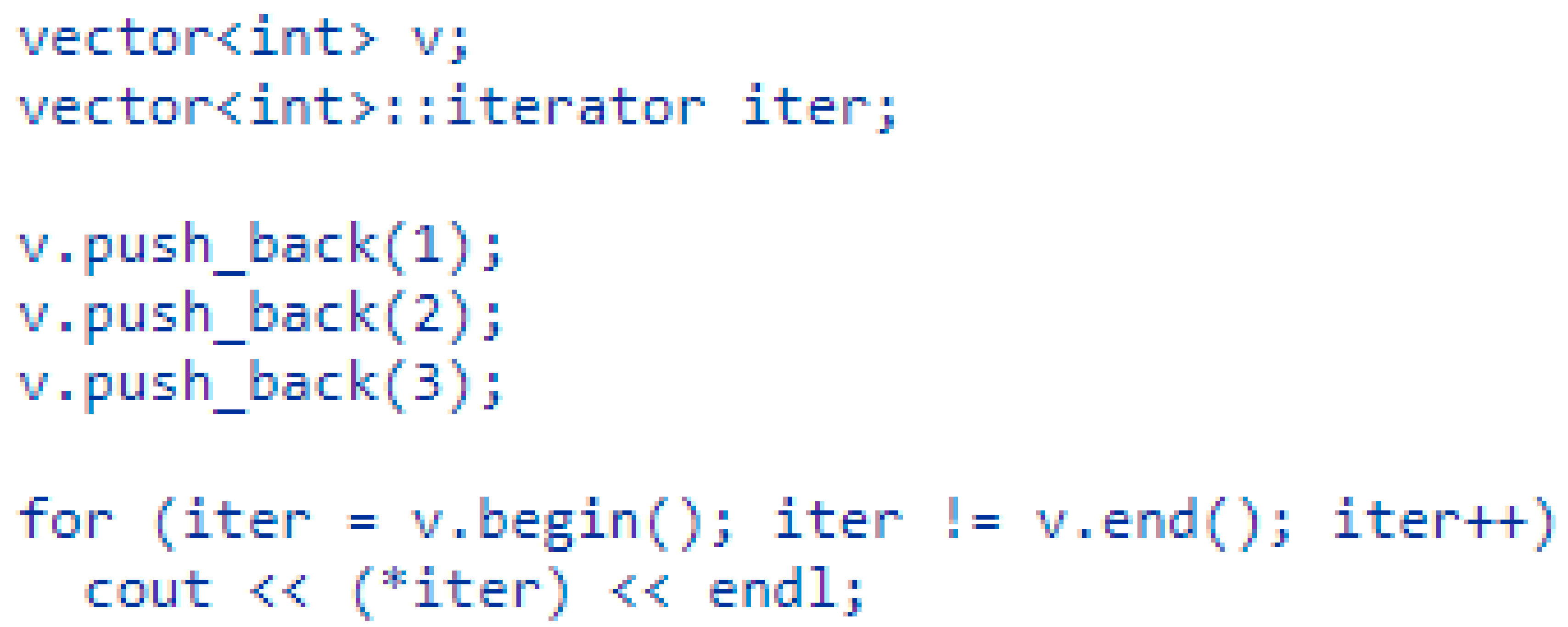
Iterators

Iterator: a pointer-like object that can be incremented with ++, dereferenced with *, and compared against another iterator with !=.

Iterators are generated by containers member functions (like vector ones), such as begin() and end(). Some containers return iterators that support only the above operations, while others return iterators that can move forward and backward, be compared with <, and so on.

The generic algorithms use iterators just as you use pointers in C to get elements from and store elements to various containers.

Iterator - example



```
vector<int> v;
vector<int>::iterator iter;

v.push_back(1);
v.push_back(2);
v.push_back(3);

for (iter = v.begin(); iter != v.end(); iter++)
  cout << (*iter) << endl;
```

In this case the iterator is used just to read data from the container (vector)

Template: use in function

A function template is a generic function that is defined on a **generic type** for which a specific type can be substituted. Compiler will generate a function for **each specific type** used.

Because types are used in the function parameters, they are also called parameterized *types*.

```
template <typename T> OR template <class T>
return-type function-name(function-parameter-list) { ...... }
```

Function Template - Example

```
/* Test Function Template (FuncationTemplate.cpp) */
#include <iostream>
using namespace std;

template <typename T>
void mySwap(T &a, T &b);
   // Swap two variables of generic type passed-by-reference
   // There is a version of swap() in <iostream>

int main() {
   int i1 = 1, i2 = 2;
   mySwap(i1, i2);   // Compiler generates mySwap(int &, int &)
   cout << "i1 is " << i1 << ", i2 is " << i2 << endl;

   char c1 = 'a', c2 = 'b';
   mySwap(c1, c2);   // Compiler generates mySwap(char &, char &)
   cout << "c1 is " << c1 << ", c2 is " << c2 << endl;

   double d1 = 1.1, d2 = 2.2;
   mySwap(d1, d2);   // Compiler generates mySwap(double &, double &)
   cout << "d1 is " << d1 << ", d2 is " << d2 << endl;

// mySwap(i1, d1);
     // error: no matching function for call to 'mySwap(int&, double&)'
     // note: candidate is:
     // note: template<class T> void mySwap(T&, T&)
}

template <typename T>
void mySwap(T &a, T &b) {
   T temp;
   temp = a;
   a = b;
   b = temp;
}
```

Function Template – Example

```cpp
/* Test function Template (TestFunctionTemplate.cpp) */
#include <iostream>
using namespace std;

template<typename T>
T abs(T value) {
   T result;    // result's type is also T
   result = (value >= 0) ? value : -value;
   return result;
}

int main() {
   int i = -5;
   cout << abs(i) << endl;

   double d = -55.5;
   cout << abs(d) << endl;

   float f = -555.5f;
   cout << abs(f) << endl;
}
```

Function Template - Overloading

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

template <typename T>
void mySwap(T &a, T &b);
   // Swap two variables of generic fundamental type

template <typename T>
void mySwap(T a[], T b[], int size);
   // Swap two arrays of generic type

template <typename T>
void print(const T * const array, int size);
   // Print an array of generic type

int main() {
   int i1 = 1, i2 = 2;
   mySwap(i1, i2);   // Compiler generates mySwap(int &, int &)
   cout << "i1 is " << i1 << ", i2 is " << i2 << endl;

   const int SIZE = 3;
   int ar1[] = {1, 2, 3}, ar2[] = {4, 5, 6};
   mySwap(ar1, ar2, SIZE);
   print(ar1, SIZE);
   print(ar2, SIZE);
}

template <typename T>
void mySwap(T &a, T &b) {
   T temp;
   temp = a;
   a = b;
   b = temp;
}
```

```
template <typename T>
void mySwap(T a[], T b[], int size) {
   T temp;
   for (int i = 0; i < size; ++i) {
      temp = a[i];
      a[i] = b[i];
      b[i] = temp;
   }
}

template <typename T>
void print(const T * const array, int size) {
   cout << "(";
   for (int i = 0; i < size; ++i) {
      cout << array[i];
      if (i < size - 1) cout << ",";
   }
   cout << ")" << endl;
}
```

Explicit Specialization

```cpp
/* Test Function Template Explicit Specialization
#include <iostream>
using namespace std;


template <typename T>
void mySwap(T &a, T &b);  // Template

template <>
void mySwap<int>(int &a, int &b);
   // Explicit Specialization for type int

int main() {
   double d1 = 1, d2 = 2;
   mySwap(d1, d2);   // use template

   int i1 = 1, i2 = 2;
   mySwap(i1, i2);   // use specialization
}
```

```cpp
template <typename T>
void mySwap(T &a, T &b) {
   cout << "Template" << endl;
   T temp;
   temp = a;
   a = b;
   b = temp;
}

template <>
void mySwap<int>(int &a, int &b) {
   cout << "Specialization" << endl;
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

Class Template

The syntax for defining a class template is as follow, where *T* is a placeholder for a type, to be provided by the user.

```
template <class T>    // OR template <typename T>
class ClassName {
   ......
}
```

The keywords class and typename (newer and more appropriate) are synonymous in the definition of template.

To use the template defined, use the syntax ClassName<actual-type>.

Class Template - Example

```
/*
 *  Test Class Template (TestClassTemplate.cpp)
 */
#include <iostream>
using namespace std;

template <typename T>
class Number {
private:
   T value;
public:
   Number(T value) { this->value = value; };
   T getValue() { return value; }
   void setValue(T value) { this->value = value; };
};

int main() {
   Number<int> i(55);
   cout << i.getValue() << endl;

   Number<double> d(55.66);
   cout << d.getValue() << endl;

   Number<char> c('a');
   cout << c.getValue() << endl;

   Number<string> s("Hello");
   cout << s.getValue() << endl;
}
```

References

https://www.tutorialspoint.com/cplusplus/index.htm

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html

https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html