# Introduction to C++ programming

ONLY SECTIONS I-VI WILL BE ASSESSED IN THE EXAM

## Content:

I. Introduction
II. A Basic C++ Program: Structure and Console I/O Operations
III. C++ Reference Variable
IV. C++ Default Function Parameters
V. C++ Function Overloading
VI. C++ Function Templates
VII. Data File and streams
VIII. C++ string data type
appendix

Check the Notes section

**Acknowledgment**: Edit to slides provided by Dr Obada Al-Khatib

# I. Introduction: C++ Programming Language

- C++ is a programming language **evolved from C**. It was **standardised by ANSI/ISO in 1998.** The latest standard revision is C++2020 (December 2020).

- C++ overcomes some of the limitations of C. In a way, C++ is a better C with some new features added.

- C++ supports **Object Oriented Methodology.** In this respect, C++ is a **completely different** language that is based on a different methodology.

# What about **C** ?

- In general, C is the best option to develop high performance applications, which efficiently utilize available hardware resources.
- However, **it may result in bulky solutions when complex data need to be processed**.

*Example:* consider writing an application that deals with different multimedia signals:

- **Speech**
- **Audio**
- **Images**
- **Video**
- **Graphics**
- **Text**
- **etc.**

The signals have different characteristics:

- Speech and audio are one-dimensional signals
- Image is two-dimensional
- Video is three-dimensional (there is temporal flow)
- Image and video are rendered through a video monitor
- Speech and audio signal are rendered through loudspeakers

How would you structure the program if you developed this application in C ?

# II. A Basic C++ Program: Structure & Console I/O Operations

```
#include <iostream>    }  includes components for
                          input/output class library
using namespace std;  ←   namespace

int main()
{
    char yourName[40];
   // I/O using C++ stream objects cin and cout
    cout << "What is your name? ";
     cin >> yourName;

    cout << "Hello " << yourName << endl;

    return 0;
}
```

# …Continued

- C++ has a number of **standard libraries** and corresponding **header files**:

  `<iostream>`

  `<fstream>` C++ header files don't have `.h` extension

  `<string>`

- Libraries place their definitions in **namespaces**

- You need to include all header files needed and specify the namespace:

```
#include <fstream>
#include <string>
using namespace std;
```

# Console Input / Output Operations

- In C, we rely on `scanf()` and `printf()` functions to implement input and output.
- In C++ `istream` and `ostream` **classes** provide input and output through **objects:**
  - `cin` - standard input stream object
  - `cout` -standard output stream object
  - `cerr` -standard error stream object
- You can use these objects along with their **member functions** and **operators** to implement I/O operations.

# Console I/O Stream Objects

- C++ streams use the **extraction operator (>>)** and the **insertion operator (<<)** to “get”/”push” variables and other objects in/out of the stream

| Name of stream object | Operator | Stream Class | Meaning |
|---|---|---|---|
| **cin** | **>>** | istream | Standard input, buffered |
| **cout** | **<<** | ostream | Standard output, buffered |
| **cerr** | **<<** | ostream | Standard error, unbuffered |

*Example1:*

```
float voltage = 3.3;

printf("Voltage = %d \n",  voltage); /* C */

cout << "Voltage = " << voltage << endl;// C++
```

*Example2:*

```
int numWins;

scanf( &numWins );  /* C */

cin >> numWins;   /C ++ */
```

Notice, that unlike scanf(), cin does not explicitly use the address of the variable numWins

# Console Output Formatting

The syntax is:

`cout << expression or manipulator << . . . ;`

- **"<<"** can deal with all fundamental data types, `int`, `float`,…
  - **`expressions`** are evaluated and their values printed
  - **`manipulators`** **are used to format the output**

    **`setprecision(n)`** : sets the number of decimal places

    **`showpoint`**: shows the decimal point even when the decimal part is 0 `setw(n)` **:** sets the width of the output field to n positions

    **`left:`** sets left justified output in the output field

    **`right:`** sets right justified output in the output field

For the above the header **iomanip** should be included

# …Continued

*Example:*

```
#include <iostream>
#include <iomanip>    //Header providing parametric manipulators:
using namespace std;

int main()
{
    double impedance = 15.454;
    double reactance = 235.87;
    double admittance = 6542.8908;

    cout << fixed << showpoint;
    cout << setprecision(2) << impedance << endl;
    cout << reactance << endl << admittance << endl;

    return 0;
}
```

**Output:**

```
15.45

235.87

6542.89
```

# Console input

- `cin` has access to **operators** and **member functions** that can be used to **extract data from standard input device**

```
        char myName[40];
        cin >> myName;
```

- Extraction operator **>>** takes two operands operands

```
    left_side_operand >>right_side_operand;
```

The *`left_side_operand`* must be of the input stream object

- You can read more than one value by using several extraction operators

```
    char myName[40];
    int myTaxFileNumber;
    cin >> myName >> myTaxFileNumber;
```

The extraction operator
skips whitespace characters
when scanning next input

# Example

```
#include <iostream>

#include <iomanip>

 using namespace std;

int main()
{
    char myName[20];
    int myTaxFileNum;

    cout << “Input your  name and tax file num:”;
   cin>>setw(20);    //  limit input to 19 characters

   cin >> myName >> myTaxFileNum;
   cout << “Your name is “ << myName << “ and your TFN is “ << myTaxFileNum
   << endl;

   return 0;
}
```

# The operator >> and white spaces

The stream extraction operator >> does not read white space characters ( *tab, end-of-line,* etc.). Unless you are reading a character,

–**leading whitespaces are ignored and removed from the buffer**

–**trailing whitespaces terminate extraction and remain in the buffer**

*Example:*

```
int num1, num2;
cout<<“Enter two numbers separated by TAB:”;
cin >> num1>>num2;

cout<<“Enter two numbers separated by TAB:”;

cin >> num1>>num2;
```

If the user input is 5*TAB*6*ENTER*7*TAB*9*ENTER*

then:

1. num1=5, *TAB* stays in the buffer
2. *TAB* is ignored, num2=6, *End_Of_line* stays in the buffer
3. *End_Of_line* is ignored, num1=7, *TAB* stays in the buffer
4. *TAB* is ignored, num2=9, *End_Of_line* stays in the buffer

# …Continued

- If you need to read an entire line into a c-string **including whitespaces and the *end-of-line* character**, you can use

  ```
  getline( char cStr[], int lengthLimit )
  ```

- If you need to read all characters (even whitespace characters) you can use the following member function:

  `get( char characterVar ):` extracts 1 byte from input stream into a char variable

*Example:*

```
char ch1, ch2;
int num1;
cin.get(ch1);
cin.get(ch2);
cin >> num1;
```

If input is:

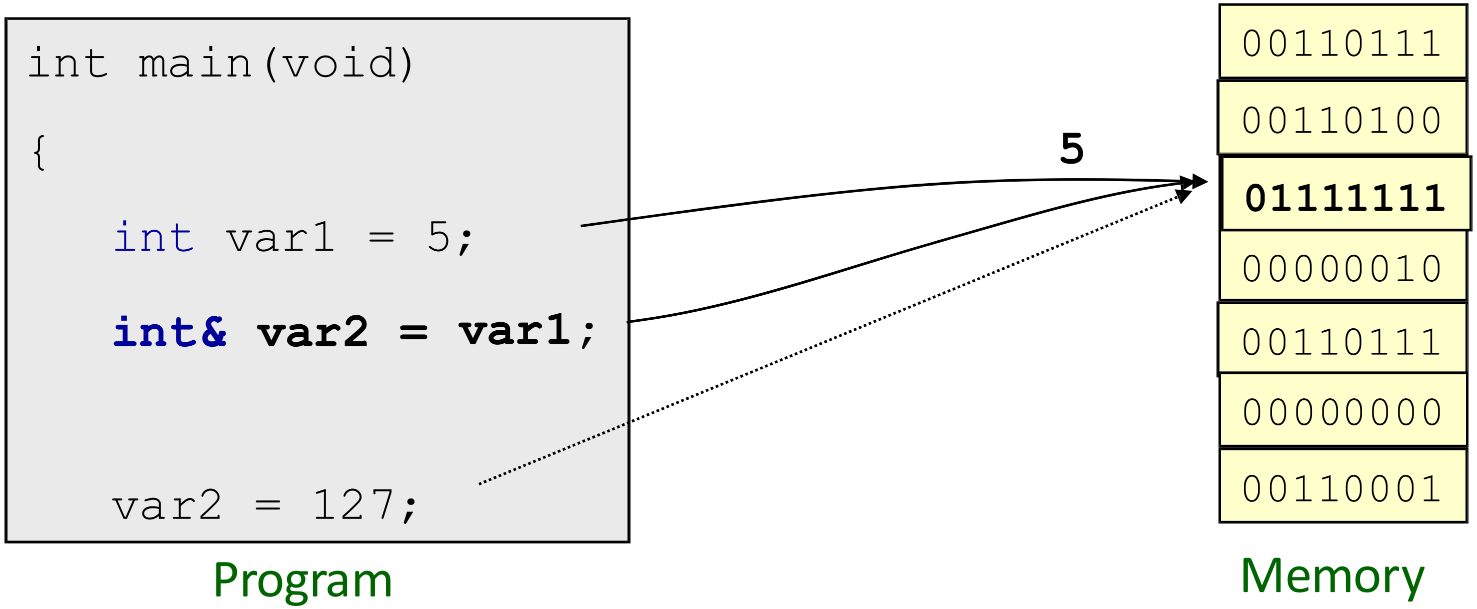
```
G 89
```

then:

```
ch1 = 'G'
ch2 = ' '
num1 = 89
```

# III. C++ Reference Variables

- **C++** introduces a new type of variables – **reference variables**
- Location of a reference variable coincides in memory with another variable.
- A reference variable is an **alias**, that is, **another name for an already existing variable**. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.



```
int main(void)
{
    int var1 = 5;
    int& var2 = var1;

    var2 = 127;
```

```
00110111
00110100
01111111
00000010
00110111
00000000
00110001
```

```
int& var3;          //compilation error, no memory location to match
int& var3 = 5;      //compilation error, 5 doesn’t have a memory location
```

# Rationale

- A more user friendly to implement **PASS by Reference**, easier  than pointers

```
 type function( type& arg );        //prototype





type function( type& arg );// definition
```

**&** indicates *passed by reference*

C++ function parameters: PASS by **Reference**

# Pass by value (C/C++)

```
int main()
{
   double examMarks[4]={24.5, 37.1, 56.4, 48.6};
   double sMark;
   . . . . .

  sMark = findSmallest(examMarks[0], examMarks[1]);

  . . .

// Function Definition
double findSmallest(double num1, double num2)
{
   if(num1<num2) return num1;
   else return num2;
}
```

**24.5** **37.1** copies

***Copies of actual parameters are passed to formal parameters***

# Pass by reference (C++) using reference variable

```
double findSmallest(double& num1, double & num2);

 int main()
 {
   double examMarks[4]={24.5, 37.1, 56.4, 48.6};
   double sMark;
   . . . . .

   sMark = findSmallest(examMarks[0], examMarks[1]);
   . . . . .
 }

 // Function
 double findSmallest(double& num1, double& num2)
 {
   if(num1<num2) return num1;
   else return num2;
 }
```

***Direct access to the actual parameters because they coincide in memory with formal parameters***

# Pass by reference (C/C++)
# Using Pointers

```
int main()
{
  double exMarks[4]={24.5, 37.1, 56.4, 48.6};
  double sMark;
  . . . . .

  sMark = findSmallest( &exMarks[0], &exMarks[1] );
  . . . . .
}

//-Function
double findSmallest(double* p1, double* p2)
{
  if(*p1 < *p2) return *p1;
  else return *p2;
}
```

***Direct access to the actual parameters through pointers***

# Return by reference

- When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement.
    - C++ functions can return values by reference

```
float& getMin( float vector[], int size )
{
   float& min = vector[0];
   for( int i=1; i < size; i++ )
     if( min > vector[i] ) min = vector[i];

   return min;  // min is a reference variable that coincides
                   in memory with one of the array element
 }
```

- Do not return by reference local automatic variables

```
float& wrongFunction(float vector[], int size)
{
   float min = vector[0];

   return min;
}   // min is often destroyed when the function returns, making the
          returned reference invalid, use static local variable instead
```

# Example

**Pass by Value**

```
void increment1( int m)
{
  ++m;                // increment a formal parameter
                     //  that is a copy of the actual parameter
}

int num = 10;
increment1( num );            // pass by value  (a copy)
cout << num << endl;          // displays 10
---------------------------------------------------------------
```

**Pass by reference through pointers**

```
 void increment2( int *p )
{
   ++(*p);             //increment a value pointed by p
}

 int num = 10;
 increment2( &num );            // pass the pointer
 cout << num << endl;           // displays 11
```

# Continued

```
void increment3( int& m)
{
    ++m;   //increment a formal parameter passed by reference
}
 int num = 10;
 increment3( num );      // pass num by reference
 cout << num << endl;    // displays 11
```

**Pass by reference**
**Using reference variable**

❖ **Pass by Reference** using reference variable is easier to interpret and less prone to errors than pointers.

- Use references when you can and pointers when you have to.
- Direct access to actual parameters may cause **data corruption** if a function has a bug (if a parameter is not to be changed, pass it as const).

```
bool saveRecord( ofstream& otFile, const data& record )
{
   record.itemPrice -= 10.0   //Error
   . . .
}
```

Any attempt to modify the structure `record` inside this function will be reported by the compiler and you can fix the bug then

# Quiz

```
// Function to swap two values
void swap(int& num1, int& num2);
const int C = 3;

int main()
{
  int a = 5, b = 8;
  . . . . .
  swap( a, b );
```

a = 8 b=5

```
  swap( a, 14 );
```

Compilation error. A reference argument cannot share memory with a numeric constant

```
  swap( a, C );
```

Compilation error. A reference argument cannot share memory with a symbolic constant

```
  swap( a, b + 1 );
}
```

Compilation error. A reference argument cannot share memory with an *rvalue*

# IV. Default Function Parameters

- **C**

  When a function is called, **ALL** its arguments **must be provided with actual values**

```
float getVolum( float w, float h, float d );

 vol = getVolum( 24.0, 36.5, 1.7 );
```

- **C++**

  Functions may include one or more default arguments/parameters

```
type function_name( type arg1, type argN = valueN );
```

- *A **formal parameter*** receives the **default value** if a program calls the function **without supplying a corresponding *actual parameter***

# …Continued

- Default arguments **may only** be specified in function prototypes

```
int getCode( char symbol, int key = 63027 ); // prototype

int getCode( char symbol, int key )          // definition
{
    .    .    .    .   .
}
```

*Example:* A function has the following prototype:

```
void funcExp( int x, int y, char ch = 'A', double w = 7.5 );
```

`ch, w` formal parameters have default values

```
funcExp(   12,  34, ‘D’, 15.5);  // OK
funcExp(   72,  36, ‘S’);        // OK
funcExp(   10,  54 );            // OK
funcExp(   12,  34, 15.5); //WRONG (warning). If ch is omitted, w must be omitted too
```

# illegal Default Function Arguments Declaration

```
void funcA( int x, double z = 34.9, char ch, int u = 62 );
```

This prototype is illegal because if the second parameter is default, all the following ones must have default values

```
void funcB( int length = 1, int width, int height = 1 );
```

This prototype is illegal because if the first parameter is default, all others after `length` must have default values

```
void funcC( int p, int& y = 19, double c = 34 );
```

This prototype is illegal because a constant value cannot be assigned to `y` since `y` is a reference parameter

# V. Function Overloading

**C:** Function name is **unique within its scope** ⟹ **choose different name**

```
int getMaxInt( int x, int y );
char getMaxChar( char first, char second );
double getMaxDouble( double d1, double d2 );
string getMaxString( string first, string second );
```

**C++:** Introduces a concept of **function overloading**

- Several functions can have the same name
- If several functions have the same name, **they must have different set of parameters**
- Parameters determine which function to execute

# …Continued

- C++ Overloading allows us to rewrite prototypes with the same function name as:

```
int getMax( int x, int y );
char getMax( char first, char second );
double getMax( double red, double blue );
string getMax( string first, string second );
```

  - **All lists of parameters are different**
  - The definitions (i.e. the body) of the functions can also be different

```
int mxNum = getMax(19, 4); //call int getMax( int, int );
char mxChar = getMax('A','V'); // call char getMax(char,char);
```

- Definition of overloaded functions with identical lists of parameters causes a compilation error (regardless the return type)

```
int getMax( int x, int y );
float getMax( int a, int b );
```

*Compilation error* ❌

# …Continued

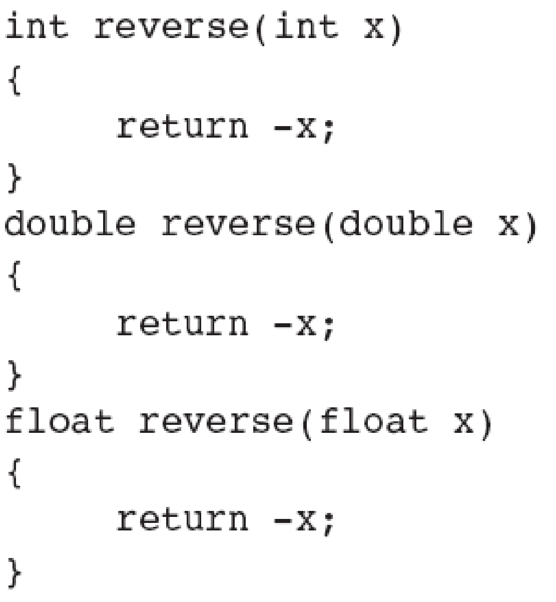
*Example:*

```
float getArea( float radius );       // for a circle
float getArea( float x, float y );   // for a rectangle
.   .   .    .
float cirArea, cirRad = 12.0;
float rectArea, width = 3.1, height=7.5;

cirArea  = getArea( cirRad );
rectArea = getArea( width, height );

// function definitions
float getArea( float radius )
{ return (3.141* radius* radius); }

float getArea( float x, float y )
{ return (x * y); }
```

# VI. C++ Function Template

- Function overloading is an efficient approach when **definitions of the overloaded functions are different.**
- **If NOT, there is a scope for enhancement.**
- *Example:* Create a function that inverses the sign of a number <u>of any type</u> (`int, float, double,`…)
  - **Naïve Solution**

Function overloading - define several functions to deal with all required types



```
int reverse(int x)
{
     return -x;
}
double reverse(double x)
{
     return -x;
}
float reverse(float x)
{
     return -x;
}
```

- These overloaded functions follow the same "pattern"
- Ideally, **you could create just one function that could be used as a template**

# …Continued

- A better solution:

```
template < typename T >
T reverse ( T x )
{
return -x;
}
```

The above template specifies a **family of functions.** The type T of the function argument is left open as a **template parameter**

- When you call a function template, the compiler:

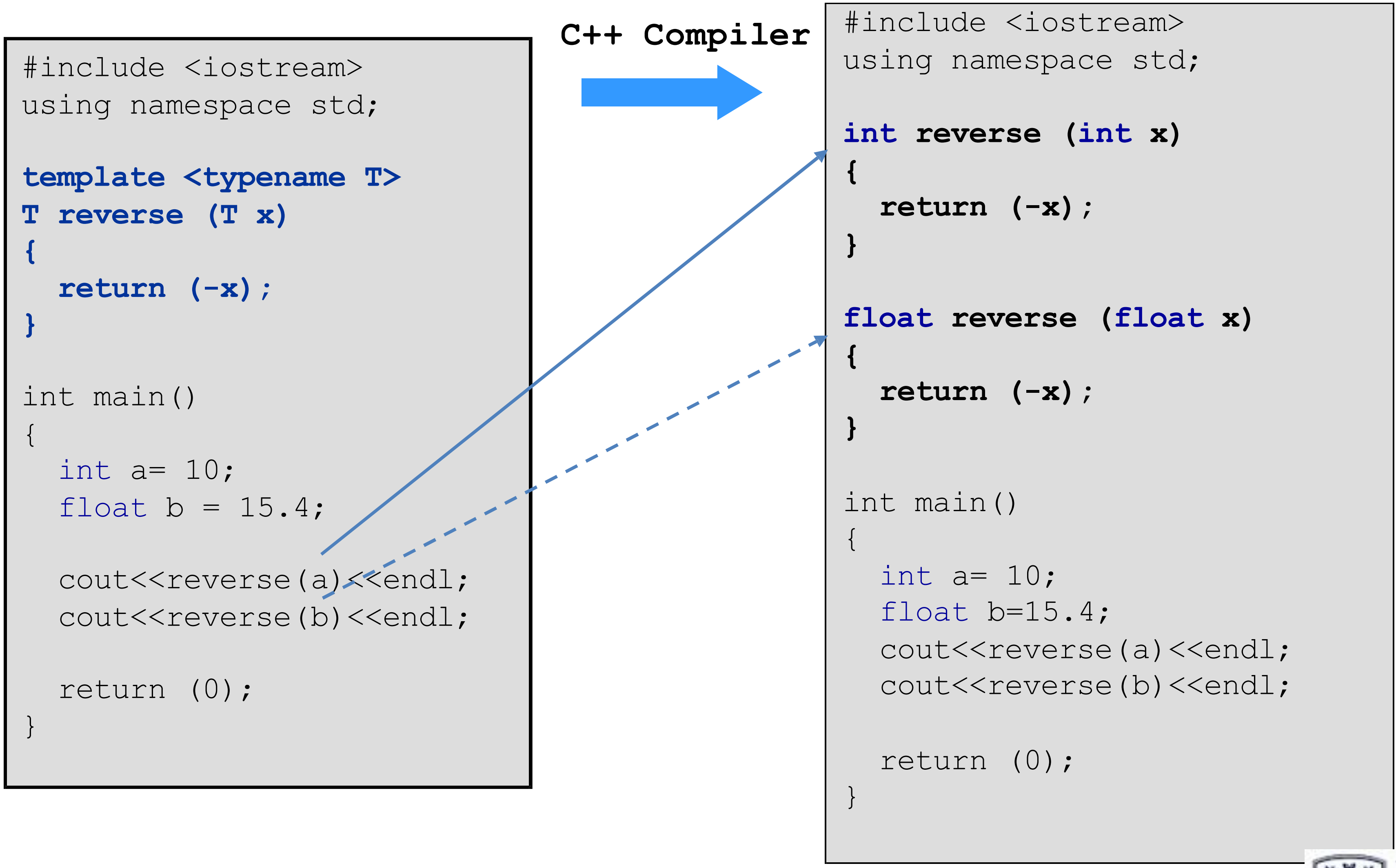
1. Determine **the type of the actual argument passed**

```
double amount = -9.86;

amount = reverse( amount );
```

2. **Generate a definition for the needed function depending on the data type of the actual arguments passed. For the above statement, the following definition will be generated**

```
double reverse(double x)
 {
      return –x;
  }
```

# …Continued



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

template <typename T>
T reverse (T x)
{
  return (-x);
}

int main()
{
  int a= 10;
  float b = 15.4;

  cout<<reverse(a)<<endl;
  cout<<reverse(b)<<endl;

  return (0);
}
```

C++ Compiler

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

int reverse (int x)
{
  return (-x);
}

float reverse (float x)
{
  return (-x);
}

int main()
{
  int a= 10;
  float b=15.4;
  cout<<reverse(a)<<endl;
  cout<<reverse(b)<<endl;

  return (0);
}
```

# Quiz

Given the function definition:

```
double findSmallest(double v1, double v2)
{
  if( v1 < v2) return v1;
  else return v2;
}
```

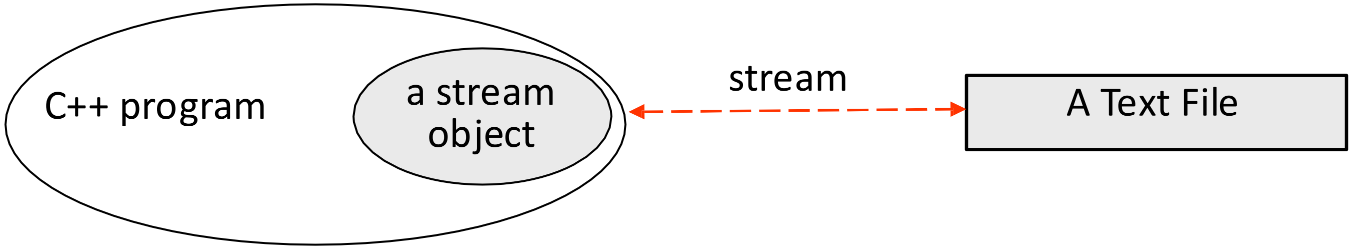
What would be the most appropriate solution to deal with other data types?

**Answer:**

```
template < typename T >
T findSmallest(T v1, T v2)
{
  if( v1 < v2)
  return v1;
  else return
  v2;
}
```

# VII. Data Files and Streams

- A variable declared in a C++ program is a part of this program.
- A file is an external collection of data that is not part of a C++ program.



- C++ must provide a linkage between the external file and its usage in the program.
- The linkage is provided by Stream Objects

# Major Steps for Creating a Stream

1. Include the header file [`I/O`]`fstream`

```
#include <fstream>
```

The library defines stream types:

**`ifstream`** – an input file stream
**`ofstream`** – an output file stream
**`fstream`** – an in/out file stream

and stream objects:

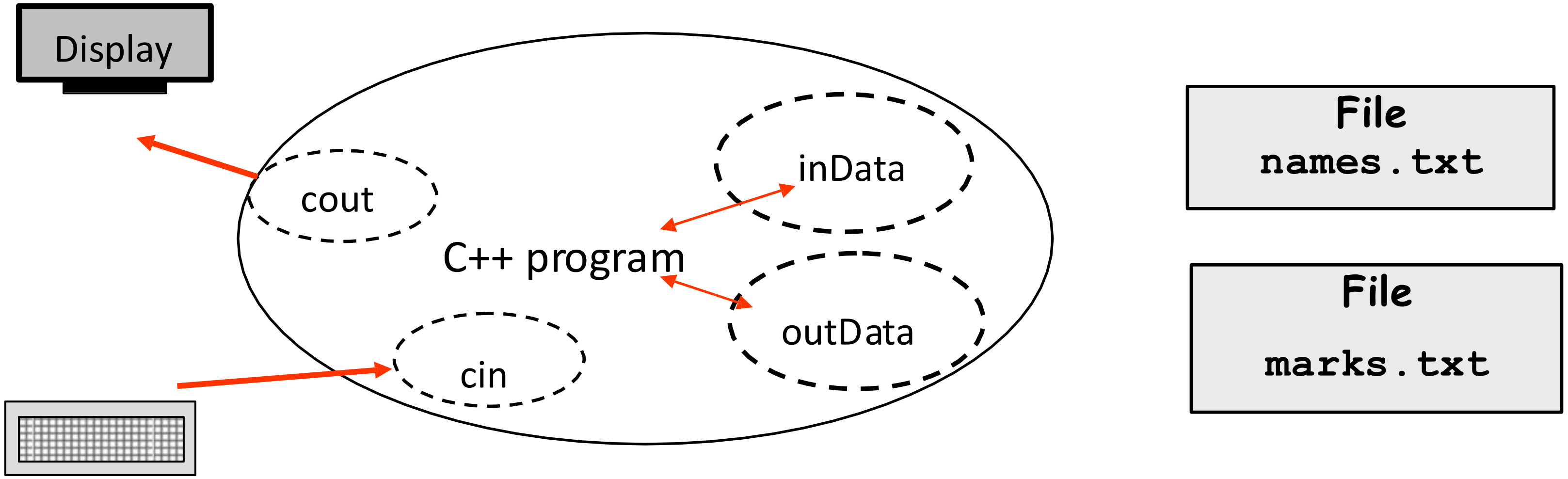
**`cin`** – standard input stream (buffered)
**`cout`** – standard output stream (buffered)
**`cerr`** – standard error stream (not buffered)

# Major Steps for Creating a Stream

2. Declare file stream objects

```
ifstream inData;
ofstream outData;
```

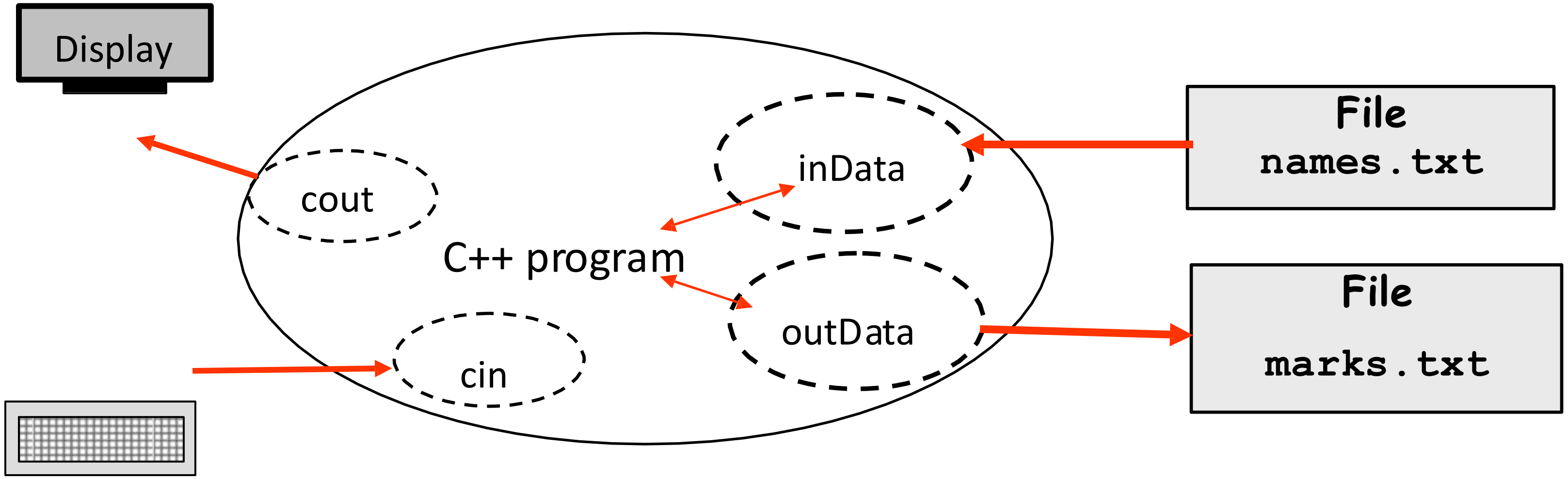
( `cin, cout` and `cerr` are already declared )



# Major Steps for Creating a Stream

3. Associate the file streams with physical files – Open Files.

```
inData.open( "names.txt" );
outData.open( "marks.txt" );
```

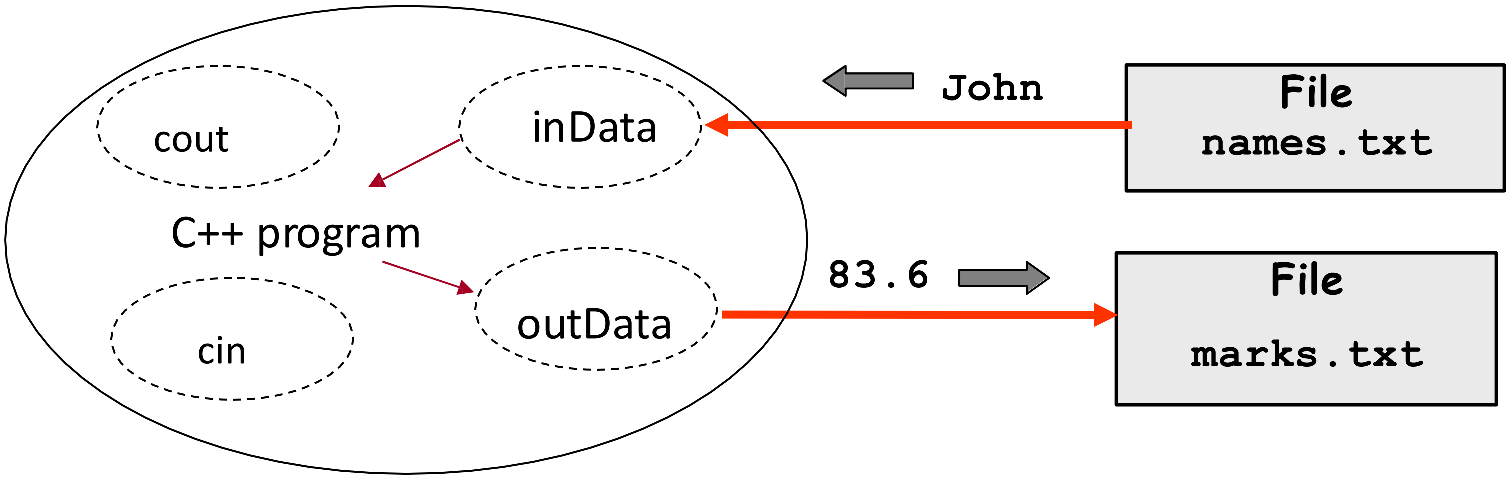


# Major Steps for Creating a Stream

4. Use >>(stream extraction) and << (stream insertion) with the file stream variables

```
inData >> firstName;
outData << examMark;
```

you can also use `put()`, `get()` and `getline()`

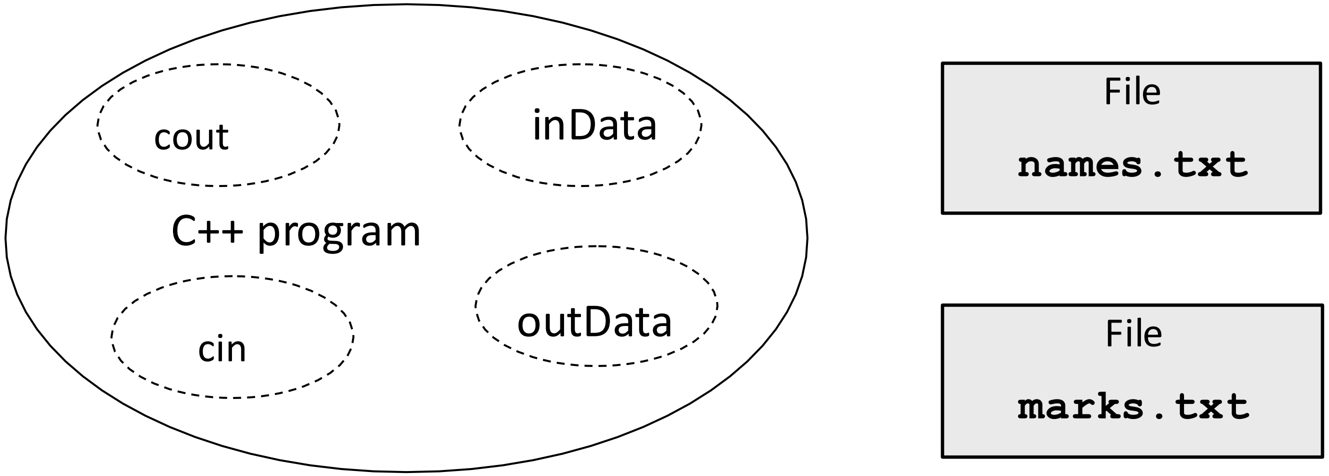


# Major Steps for Creating a Stream

## 5. Disconnect from the physical files – Close the Files

```
inData.close();
outData.close();
```

- Files remain in the computer file system when they are closed
- File stream objects remain in the program and can be reopened



# Example

```
1   #include <iostream>
    #include <fstream>
    using namespace std;
    int main()
    {
2       ifstream inData;          //declare an input file stream
        ofstream outData;         //declare an output file stream
        string firstName, lastName;

3       inData.open("names.txt");     // open a stream for input
        outData.open("marks.txt");    // open a stream for output

4       inData>>firstName>>lastName;
        outData<<85.6;

5       inData.close();    // Close the input file stream
        outData.close();   // Close the output file stream
        return 0;
    }
```

# File Opening Modes

- Function open

```
void open (const char filename[], openmode mode);
```

| | | |
|---|---|---|
| **ios::in** | Open file for reading | |
| **ios::out** | Open file for writing | |
| **ios::app** | Every output is appended at the end of file | If the file does not exist, an empty file is created |
| **ios::trunc** | If the file already existed it is erased | |

**Ios::binary**: open a binary file

*Examples:*

open a text file for output. If it does not exist, it must be created. If it already exists, the existing content must be erased

To combine modes

```
outFile.open("report.txt", ios::out | ios::trunc);
```

open a binary file for input

```
inData.open("report.bin", ios::binary | ios::in );
```

open a binary file for output in the append mode

```
outFile.open( "temp.dat", ios::binary | ios::app );
```

# Open File Errors

Do not assume that a file stream is always opened successfully

- Incorrect file name: inFile.open("names.txl");

- Incorrect file opening mode

```
ifstream inFile;
inFile.open( "names.txt", iso::trunc );
```

- Not enough room on the hard drive

- Hardware failure

You always must check the status of a stream after using `open`

# Open File Errors

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream inData;       //declare an input file stream
    char fileName[] = "exams.txt";

    inData.open( fileName );       // open input file
    if( !inData )    // check if open failed
    {
       cerr << "Error opening :“ << fileName;
       return -1; // exit with an error code
    }
    inData >> lastName >> mark;

    inData.close();         // Close the input file
    return 0;
}
```

# Stream Insertion operator failure

- What if the stream insertion operator << fails?

```
outFile << "Date :" << date << endl;
if( !outFile )  // check for << failure
{
   cerr << "Output stream failure" << endl;
   outFile.close();
   return -2;
}
```

# File Streams as function parameters

- File stream objects must be passed to functions by **reference**, not by value

```
int report( ifstream& inFile )   // correct
int report( ifstream  inFile )   // incorrect
```

- If you pass stream objects by value, the C++ compiler will not complain, but...
- Errors will start happening when you run the program

# File Streams as Function Arguments

```
bool writeReportHeader( ofstream& outFile, document report)
{
   if( !outFile )  return false; // check the stream

   outFile << "Report date :" << report.date << endl;
   if( !outFile )  return false;

   outFile << report.title << endl;
   if( !outFile ) return false;


   return true;
}

//--------- function call ------------
bool status = writeReportHeader( reportFile, saleReport );
if( status == false ) cerr << "WriteReport failure“<<endl;
```

# File Reading Errors

- When reading data from a text file, various errors may happen
  - The program may not have data to read as it hits the end of file
  - Data may be invalid (alphabetic characters instead of digit characters, etc)
  - Data may not be physically accessed from a hard drive due to its damage
- Is this a generic solution for all possible problems?

```
 if( !inData )
{
   Error Recovery Action
}
```

# Reading from a File

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

int main()
{
   double nextNumber, averTemp = 0.0;
   int total = 0;
   ifstream inFile;

   inFile.open("may_2016.txt");
   if(!inFile) {  //check for open errors
      cerr << "File opening error" << endl;
      return -1;
   }

   while( inFile >> nextNumber ) { //while reading is correct
      averTemp += nextNumber; total++;
   }
   cout<<"The average temperature is: "<< averTemp/total;

   return 0;
}
```

# File Reading Errors

- What if an alphabetic character was mistakenly written to the file ?

```
24.56 25.02 24.04 G 23.15 22.47 22.13 21.84
```

```
while( inFile>>nextNumber )   //while reading is correct
  averTemp += nextNumber;
```

The error will be detected, but we can assume that all file data have been processed

- Can this solution help ?

```
while( !inFile.eof() ) {    //while not end-of-file
  inFile>>nextNumber;
  averTemp += nextNumber;
}
```

Even worse -an infinite loop in the case of a wrong character

# File Reading Errors

- A comprehensive error checking and appropriate error recovery need to be implemented.
- C++ **provides three status flags and four functions** to detect possible errors

1.The flag `eof` indicates that the end of file is reached

```
if( inData.eof() )  {    Error recovery action }
```

2. The flag `fail` indicates a failure due to invalid data

```
if( inData.fail() ) {    Error recovery action }
```

3. The flag `bad` indicates a hardware problem

```
if( inData.bad() ) {     Error recovery action }
```

4.The function `good()` returns `true` if no any error has been detected

# File Reading Errors

```
int readData( ifstream& inFile, double& averTemp )
{
   double nextNumber;
   int total = 0;

   averTemp = 0.0;

   while( inFile>>nextNumber ) {  //while reading is correct
      averTemp += nextNumber;
      total++;
   }
   if( inFile.fail() )  return -1;   // invalid character
   if( inFile.bad() )   return -2;   // hardware failure
   if( total==0 )       return -3;   // an empty file

   averTemp /= total;

   return 0;
}
```

# Error Recovery

- Once the stream is in the error state, it will stay that way and all subsequent operations will **do nothing.**
- You have to **clear** the stream by calling `clear()` function before the stream recovers again.
- The input buffer may still contains incorrect values.
- To clean the buffer from the garbage

```
cin.ignore(BUFSIZE,‘\n’);
```

*Example:*

```
inFile >> newNumber;
if( inFile.fail() )
{
   inFile.clear();          // get back to the working state
   cin.ignore(80, ‘\n’);    // clean the buffer
}
```

# Reading Formatted Data

- A program must find a student with ID Number `573001`from a list stored in a text file containing *student's first name, last name* and *ID number.*

- File format details:

  - The first name and the last name are separated by a blank.
  - ID Number follows the last name and is separated by tab.
  - Each student Info starts from a new line.

**Example:**

```
Tom Green   345975
Peter Stone    140056
Dan Smith   573001
```

# Reading Formatted Data

```
int readData( ifstream& inFile, record_t& student )
{
   string nameFirst, nameLast, id=“”;

   do
     inFile>>nameFirst>>nameLast>>id;
   while( inFile.good() && id != "573001" );

   if(inFile.fail())  return -1;   // invalid character
   if(inFile.bad())   return -2;   // hardware failure
   if(inFile.eof()    ) return -3; // not found

   student.firstName = nameFirst;
   student.lastName = name.Last;
   student.id = id;

   return 0;
}
```

What may happen if a name has a blank character ?

# Character Input/Output

- How to read the entire input file stream including blanks, tabs and new-lines ?
- A `stream` member function that reads a character

```
char nextChar; cin.get(nextChar);
```

**or**

```
ifstream inFile; inFile.get(nextChar);
```

- You can use `put` function with output file streams

```
ofstream outFile; outFile.put(nextChar);
```

# Example

A function to copy a text in one file to another file.

```
int copyFile(ifstream& infile, ofstream& outfile)
{
   char tempChar;

   while( infile.get(tempChar) )
       outfile.put(tempChar);

   if( infile.eof() ) return 0; // copied till eof
   else return -1;
}
```

```
ifstream inFile;
ofstream outFile;
```

```
Function Call:
   status = copyFile(inFile, outFile);
   if( status != 0 ) cerr<<“File copy error<<endl;
```

# VII. C++ String Data Type

- C++ string data type is defined in the `string` C++ standard library.

A header file needs to be included

```
#include <string>
```

- A string type object declaration

```
string objectName;
```

- You do not need to specify string size because it is handled dynamically.

# String Concatenation

- Strings can be concatenated using + operator. You can use string objects, string literals and characters for concatenation.

*Example:*

```
string str1 = "Happy";
string str2 = "New Year";

string greetings = str1 + ' ' + str2 + " !";
```

Stores the string `"Happy New Year !"` into `greetings`

- One of the operands must be a string variable

```
greetings = "Happy " + "New Year";    // illegal
```

# String Comparison

- All relational operators ( `<, >, ==, <=, >=, !=` ) can be used to compare strings

*Example:*

```
string s1 = "John", s2 = "James", s3 ="Hi ";
.    .   .   .   .
if (s1 > s2)
   s3 = s3 + s1;
else
   s3 = s3 + s2;
```

Strings are compared sequentially using ASCII codes of the characters

# String Size

- After all manipulations with a string its size is automatically adjusted.

```
string city = "Sydney NSW";
cout<<city.size()<<endl;          // output is 10
```

```
city = city + " " + "2100";
cout<<city.size()<<endl;          // output is 15
```

- Unlike with c-strings you do not need to worry about the string memory allocation.

# Manipulations with strings

- A string may be inserted at a particular position into another string

```
insert( int startPosition, string inString );
```

where

`startPosition` – a position at which insert begins

`inString` – a string to be inserted

*Example:*

```
string chptTitle = "An Overview of Languages";
chptTitle.insert( 15, "Programming ");
cout << chptTitle;
```

*Prints :* `An Overview of Programming Languages`

# Manipulations with Strings

- A substring of a string can be replaced with another substring

```
replace(int startFrom, int numToErase, string subSt);
```

where:

`startFrom` – beginning of the substring to be replaced

`numToErase` – number of characters to be erased and replaced

`subSt` – a substring that replaces the erased segment

*Example:*

```
string city = "San Francisco CA";
. . . . . . .
city.replace( 4, 9, "Diego");
cout<<city;
```

Prints : `San Diego CA`

# Strings and Text Files

- You can use >> to read strings <u>without white spaces</u>

```
ifstream inFile;
string firstName, lastName;
.   .   .
inFile >> firstName >> lastName;
```

- You can use `getline()` to read a whole line into a string

*another overloaded version of getline()*

```
ifstream inFile;
string lineOfText;
.   .   .
while( getline(inFile, lineOfText) )  // while in the read state
{
        // use string manipulation functions to process the line
}
```

APPENDIX NEXT

# Common Programming Errors

- Header file `<fstream>` is not included
  - `#include <fstream>`
- File stream is not closed when you finish working with it.
- Opening file for output in default mode will erase the previous file content. Use `append` mode.
  - `outFile.open( outFileName, ios::app );`
- Opening of a file that already has been opened.
- Reading from a file that has been opened for output.
- Not all possible errors with file operations are detected and properly processed.

# The operator >> and whitespaces

- It is convenient to use:
  - `get()` to process each character entered by the user
  - `>>` to process each word, or a number
  - `getline()` to process lines of characters

- You need to take buffering into account when you use `>>` and input stream functions ( `get()` and `getline()`) in the same program

  –Carefully analyze your program I/O and content of the input buffer after each operation

  –If the input buffer is expected to be empty before new user input takes place, but there is a risk of buffer contamination due to improper previous user input, clean the buffer

*Example:* clear the buffer until the *end-of-line* or up to 100 characters whichever comes first

```
cin.ignore( 100, '\n' );
```

# C99, C11, C++ inline functions

- C uses the stack to push function parameters and return values. As a result, repeated calls of small functions may have performance cost.

```
for(int i=0; i<SIZE; i++)
code[i]= en~~codeData( data[i], key );~~
```

Replaces the function call

- Keyword inline can make function calls execute faster
  - The compiler replaces a function call with the function body code
  - Eliminates the need to push arguments into the stack and return values

```
inline int encodeData( int data, int key );
.    .    .    .
inline int encodeData( int data, int key );
{
      return (data<<key);
   }
```

```
code[i] = data[i]<<key;
```

Note: Modern compilers can automatically make decisions to inline functions if this can improve performance

# Dynamic memory allocation

- C

```
malloc(), calloc(), realloc() and free()

int *iPtr;
iPtr = (int*)malloc( 25*sizeof(int) );
```

- C++ introduces **new** and **delete**

```
int *iPtr;
iPtr = new int[25];   // create a dynamic array
.   .   .   .
delete []iPtr;        // free the memory block
```

There is no any equivalent of `realloc` in C++ to manipulate arrays.

# C++ dynamic memory allocation

```
float *arrayFt;

arrayFt = new float[10];     // allocate an array of float

for(int i = 0; i < 10; i++)
  arrayFt[i] = 15.0;          // initialize dynamic array

delete [] arrayFt;            // free the allocated memory
```

**[ ]** is required to free a dynamic array

---

```
date *newDate;

newDate = new date;          // allocate one structure

newDate->day = 3;

delete newDate;              // free the allocated memory
```

**[ ]** is not required to free a single variable