Data Structures and Algorithms

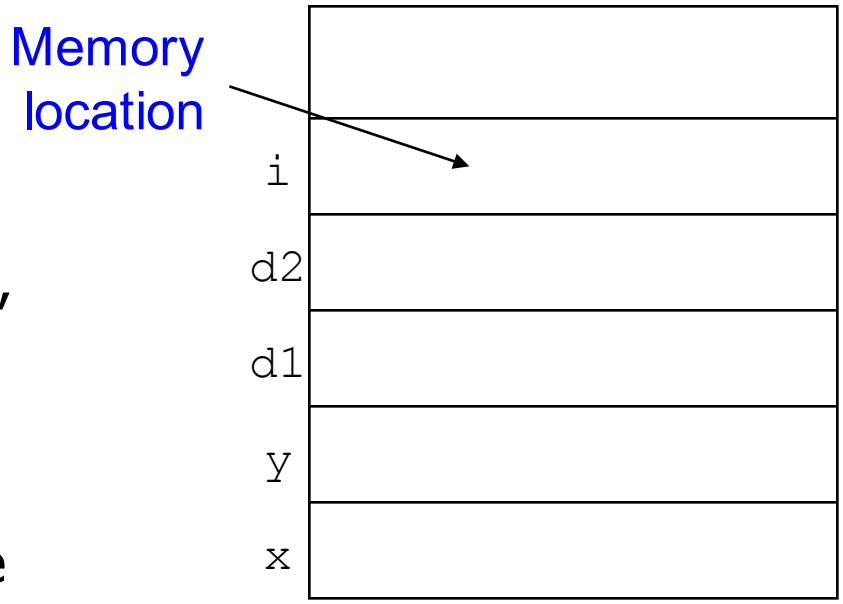
Lecture1 : C++ Review

Outline

- C++ basic features
  - Programming paradigm and statement syntax
- Class definitions
  - Data members, methods, constructor, destructor
  - Pointers, arrays, and strings
  - Parameter passing in functions
- I/O streams
  - An example on file copy

Functions & Memory

- Every function needs a place to store its local variables. Collectively, this storage is called the *stack*
- This storage (memory aka "RAM"), is a series of storage spaces and their numerical addresses
- Instead of using raw addresses, we use variables to attach a name to an address
- All of the data/variables for a particular function call are located in a *stack frame*

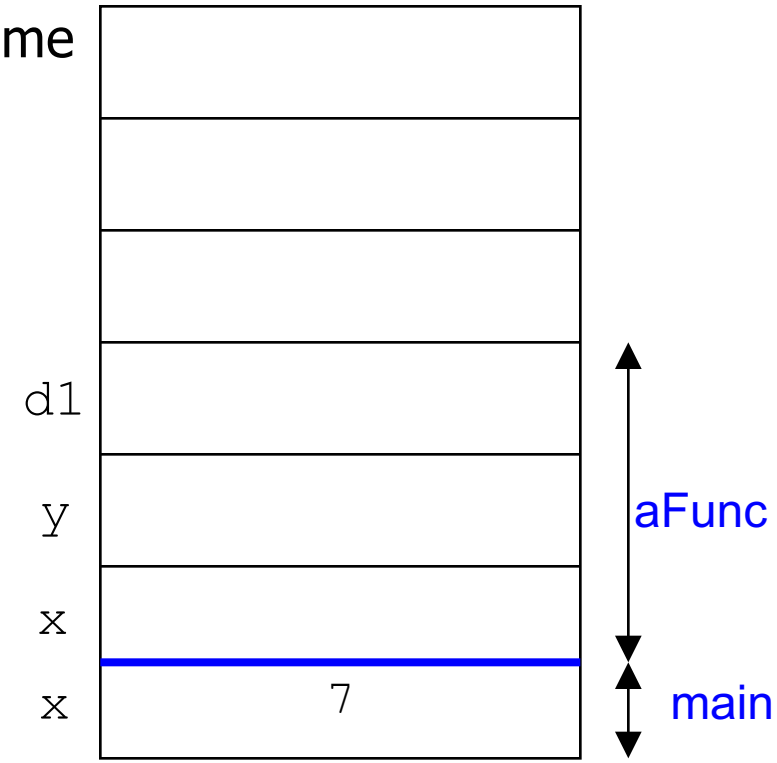



```
void aFunc(int x, int y)
{
   double d1, d2;
   int i;
}
```

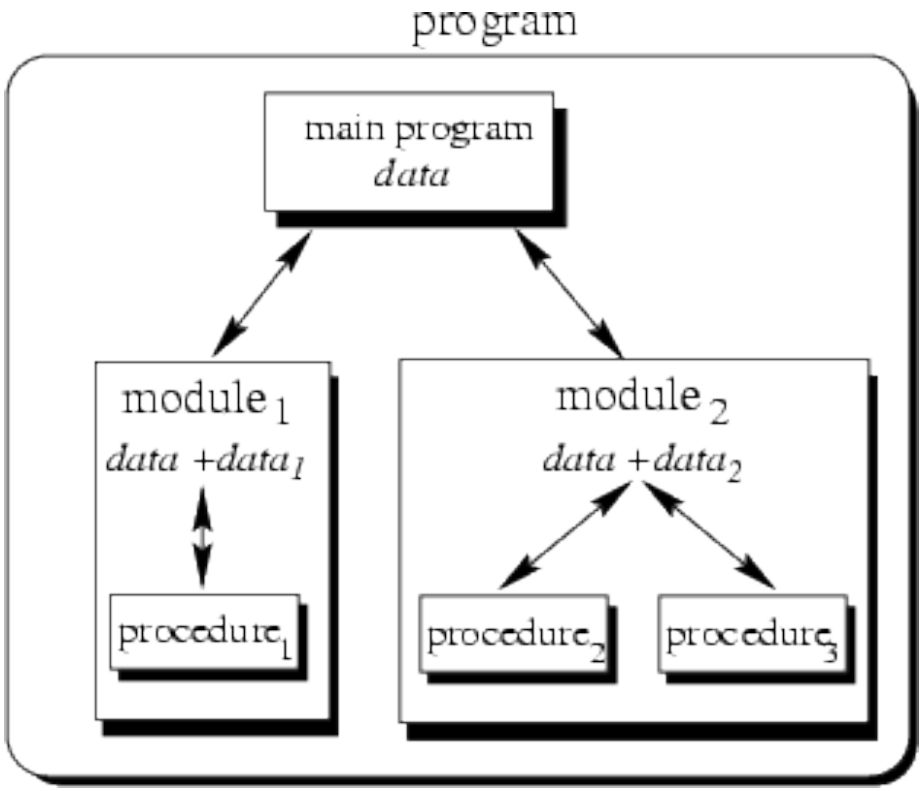
Functions & Memory (cont)

- When a function is called, a new stack frame is set aside
- Parameters and return values are passed *by copy* (ie, they're copied into and out of the stack frame)
- When a function finishes, its stack frame is reclaimed

```
void aFunc(int x, int y) {
  double d1 = x + y;
}
int main() {
   int x = 7;
   aFunc(1, 2);
   aFunc(2, 3);
   return 0;
}
```



Programming Paradigm: Modular Concept

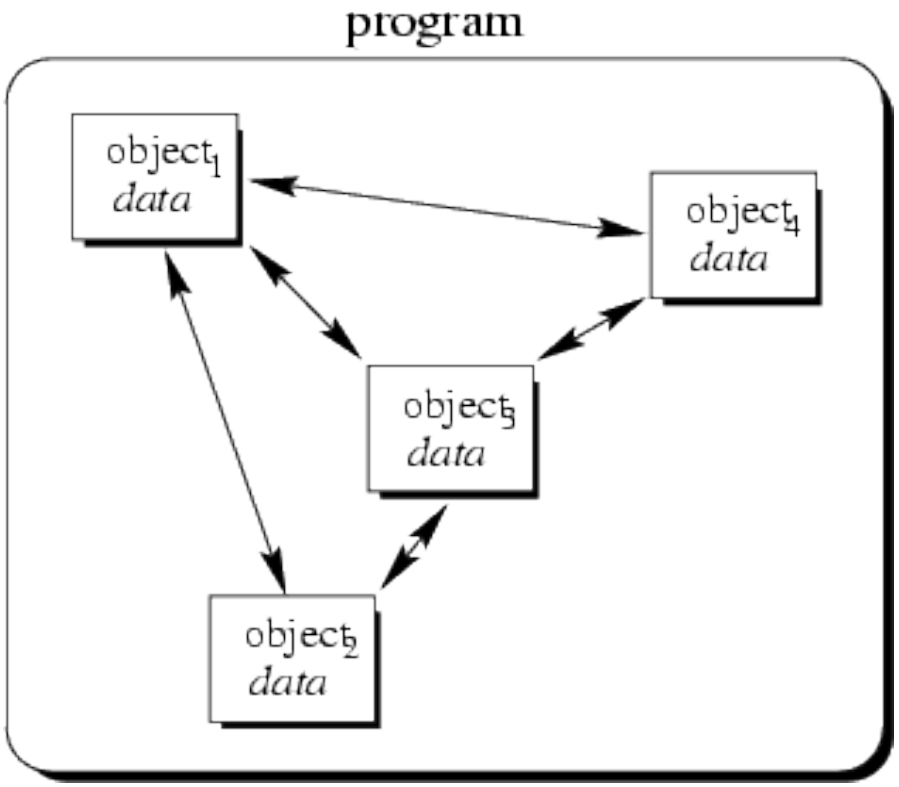


- The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters

Modular Concept - Problems

- **Decoupled Data and Operations**
  - The resulting module structure is oriented on the operations rather than the actual data
  - The defined operations specify the data to be used.

Object-Oriented Concept (C++)



- Objects of the program interact by sending messages to each other

Basic C++

- Inherit all C syntax
  - Primitive data types
    - Supported data types: `int`, `long`, `short`, `float`, `double`, `char`, `bool`, and `enum`
    - The size of data types is platform-dependent
  - Basic expression syntax
    - Defining the usual arithmetic and logical operations such as `+`, `-`, `/`, `%`, `*`, `&&`, `!`, and `||`
    - Defining bit-wise operations, such as `&`, `|`, and `~`
  - Basic statement syntax
    - `If-else`, `for`, `while`, and `do-while`

Basic C++ (cont)

- Add a new comment mark
  - // For 1 line comment
  - /*… */ for a group of line comment
- New data type
  - Reference data type “&”. Much likes pointer

```
int ix; /* ix is "real" variable */
int & rx = ix; /* rx is "alias" for ix */
ix = 1; /* also rx == 1 */
rx = 2; /* also ix == 2 */
```

- *const* support for constant declaration, just likes C

Class Definitions

- A C++ class consists of *data members* and *methods (member functions).*

```
class IntCell
{
   public:
        IntCell( int initialValue = 0 )
           { storedValue = initialValue; }

        int read( ) const
           { return storedValue;}
        void write( int x )
           { storedValue = x; }
   private:
        int storedValue;
}
```

Constructor: automatically called when an instance of the class is declared.

Member functions

Indicates that the member's invocation does not change any of the data members.

Data member(s)

Information Hiding in C++

- Two labels: *public* and *private*
  - Determine visibility of class members
  - A member that is *public* may be accessed by any method in any class
  - A member that is *private* may only be accessed by methods in its class
- Information hiding
  - Data members are declared *private,* thus restricting access to internal details of the class
  - Methods intended for general use are made *public*

Constructors

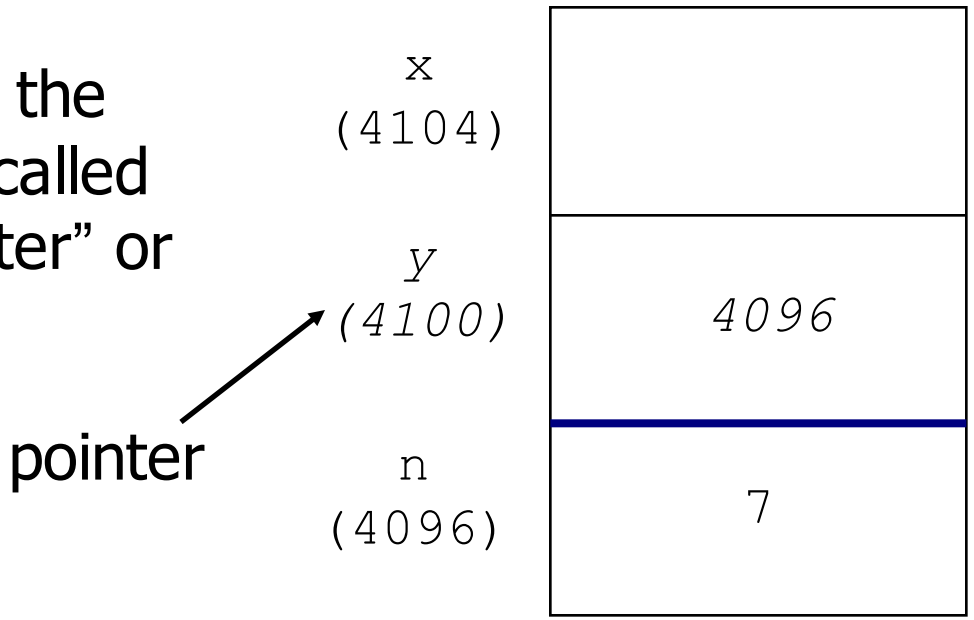
- A *constructor* is a special method that describes how an instance of the class (called *object*) is constructed
- Whenever an instance of the class is created, its constructor is called.
- C++ provides a *default constructor* for each class, which is a constructor with no parameters. But, one can define multiple constructors for the same class, and may even redefine the default constructor

Destructor

- A *destructor* is called when an object is deleted either implicitly, or explicitly (using the *delete* operation)
  - The destructor is called whenever an object goes out of scope or is subjected to a *delete*.
  - Typically, the destructor is used to free up any resources that were allocated during the use of the object
- C++ provides a *default destructor* for each class
  - The default simply applies the destructor on each data member.
  - One can redefine the destructor of a class.
  - A C++ class can have only one destructor.
- A C++ class can have only one destructor

Pointers

- A *pointer* is a variable which contains the address of another variable
- Accessing the data at the contained address is called "dereferencing a pointer" or "following a pointer"

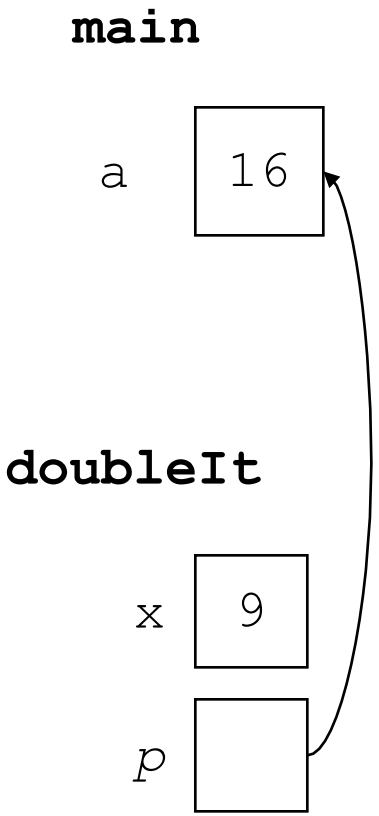


A Pointer Example

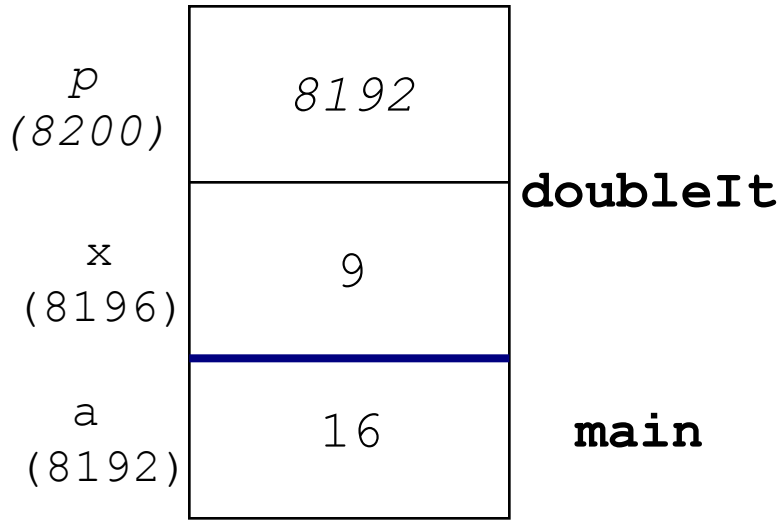
The code

```
void doubleIt(int x,
              int * p)
{
    *p = 2 * x;
}
int main(int argc,
const char * argv[])
{
    int a = 16;
    doubleIt(9, &a);
    return 0;
}
```

Box diagram



Memory Layout



a gets 18

Interface and Implementation

- In C++ it is more common to separate the class *interface* from its *implementation*.
- The *interface* lists the class and its members (data and functions).
- The *implementation* provides implementations of the functions.

```
class IntCell{
    public:
        IntCell( int initialValue = 0 );
        int read( ) const;
        void write( int x );
    private:
        int storedValue;}
```

IntCell.h

```
IntCell::IntCell( int initialValue )
    { storedValue = initialValue; }
int IntCell::read( ) const
    { return storedValue; }

void IntCell::write( )
    { storedValue = x; }
```

IntCell.cpp

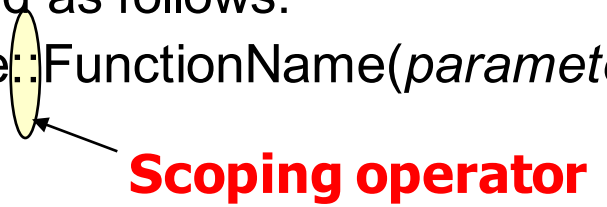
The interface is typically placed in a file that ends with *.h.* The member functions are defined as:

ReturnType FunctionName(*parameterList*);

The implementation file typically ends with *.cpp, .cc,* or *.C.* The member functions are defined as follows:

ReturnType ClassName::FunctionName(*parameterList)*

{ …… }



Object Pointer Declaration

- Declaration

```
IntCell * p;
//defines a pointer to an object of class
//IntCell
```

  - The * indicates that *p* is a pointer variable; it is allowed to point at an IntCell object.
  - The *value* of *p* is the address of the object that it points at
  - *P* is uninitialized at this point
  - The use of uninitialized pointers typically crashes programs

Dereferencing Pointers

- Dynamic object creation

  p = *new* IntCell;

  In C++ *new* returns a pointer to the newly created object.

- Garbage collection
  - C++ does not have garbage collection
  - When an object that **is allocated by *new*** is no longer referenced, the *delete* operation must be applied to the object

    delete *p;*

Dereferencing Pointers (cont)

- Using a pointer

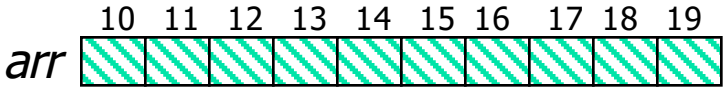
We can get the value of the object pointed at by a pointer either by using operator *, or by using operator ->

```
IntCell a;
int b;

……

a =*p;
//variable a gets the value of object pointed at by p
b = p->read( );
//the value of the data member stored.
// Value of the object pointed at by p is assigned to b
```

Array Declaration

- An *array* is a collection of objects with same type stored consecutively in memory
- Declaring an array

```
IntCell arr[10]; //an array consisting of 10 IntCell
objects
```



  - The size of the array must be known at compile time.
  - *arr* actually is a constant pointer. The value of *arr* **cannot** be changed.

```
IntCell * p = new IntCell[10];
arr = p; // invalid
```

  - The *(i+1)*-st object in the array *arr* can be accessed either by using *arr[i],* or by **(arr+i).*
- There is no index range checking for arrays in C++
- Cannot be copied with =

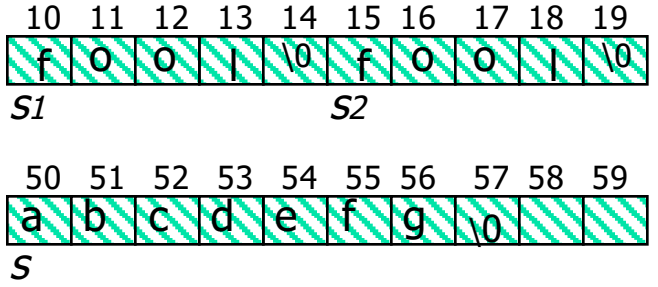
Strings

- Built-in C-style strings are implemented as an array of characters.
- Each string ends with the special null-terminator '\0'.
- *strcpy*: used to copy strings

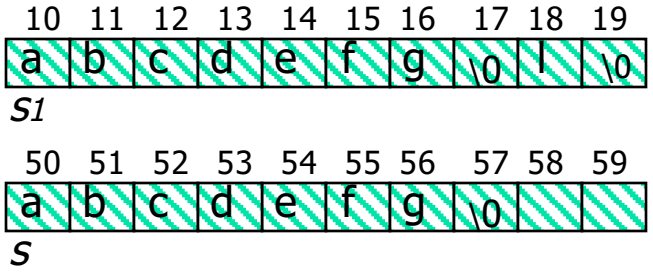
  strcmp: used to compare strings

  strlen: used to determine the length of strings
- Individual characters can be accessed by the array indexing operator

```
char s1[] = "fool";

char s2[] = "fool";

char s[]="abcdefg";
```




```
strcpy(s1, s);
//copy s to s1
//(s1 must have enough size)
//including \0
```



Function Call by Value

```
int f(int x) { cout << "value of x = " << x << endl;
               x = 4; }

    main() { int v = 5;
             f(v);
             cout << "value of v = " << v << endl;}
```

Output: Value of x = 5

Value of v = 5

- When a variable *v* is passed *by value* to a function *f*, its value is copied to the corresponding variable *x* in *f*
- Any changes to the value of *x* does NOT affect the value of *v*
- Call by value is the default mechanism for parameter passing in C++

Function Call by Reference

```
int f(int &x) { cout << "value of x = " << x << endl;
                x = 4; }

     main() { int v = 5;
              f(v);
              cout << "value of v = " << v << endl;}
```

Output: Value of x = 5

Value of v = 4

- When a variable *v* is passed *by reference* to a parameter *x* of function *f*, *v* and the corresponding parameter *x* refer to the same variable
- Any changes to the value of *x* DOES affect the value of *v*

Function Call by Constant Reference

```
int f( const int &x ) { cout << "value of x = " << x << endl;
                        x = 4;  // invalid }

main() { int v = 5;
         f(v);
         cout << "value of v = " << v << endl;}
```

- Passing variable *v by constant reference* to parameter *x* of *f* will **NOT** allow any change to the value of *x.*
- It is appropriate for passing large objects that should not be changed by the called function.

Usage of Parameter Passing

- *Call by value* is appropriate for small objects that should not be changed by the function
- *Call by constant reference* is appropriate for large objects that should not be changed by the function
- *Call by reference* is appropriate for all objects that may be changed by the function

Reference Variables

- Reference and constant reference variables are commonly used for parameter passing to a function
- They can also be used as local variables or as class data members
- A reference (or constant reference) variable serves as an alternative name for an object.

```
int m = 10;
int & j = m;
cout <<"value of m = " << m << endl;
//value of m printed is 10
j = 18;
cout << "value of m = " << m << endl;
//value of m printed is 18
```

Reference Variables (cont)

- A reference variable is different from a pointer
  - A pointer need NOT be initialized while defining, but a reference variable should always refer to some other object.

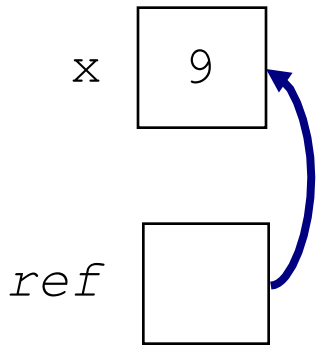
```
int m = 10;

int * p;
p = & m; //valid

int & j = m; //valid
int & k; //compilation error
```

References (Summary)

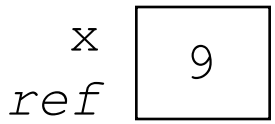
References are an additional name to an existing memory location

If we wanted something called "`ref`" to refer to a variable `x`:

Pointer:



Reference:



Pointer vs. Reference

- A pointer can be assigned a new value to point at a different object, but a reference variable always refers to the same object. Assigning a reference variable with a new value actually changes the value of the referred object.

```
int * p;
int m = 10;
int & j = m; //valid
p = &m; //p now points at m
int n = 12;
j = n;
// the value of m is set to 12. But j still refers to m, not to n.
cout << "value of m = " << m <<endl;
//value of m printed is 12
n = 36;
Cout << "value of j = " << j << endl;
//value of j printed is 12
p = &n;
```

Pointer vs. Reference (cont)

- A *constant reference variable* *v* refers to an object whose value cannot be changed through *v.*

```
int m = 8;
const int & j = m;
m = 16; //valid
j = 20; //compilation error
```

Return by reference

```
int & foo(int &b){
  return b;
}

main(){
  int j;
  int a = 5;

  j=foo(a);  //j is 5
  j=3;  // a is still 5

  foo(a) = 10; //a is now 10
}
```

```
const int & foo(int &b){
  return b;
}

main(){
  int j;
  int a = 5;

  j=foo(a);  //j is 5
  j=3;  // a is still 5

  foo(a)= 10;  // invalid
}
```

A C++ Example

- point.h

```
class Point {
 private:
    int _x, _y;        // point coordinates

  public:              // begin interface section
    void setX(const int val);
    void setY(const int val);
    int getX() { return _x; }
    int getY() { return _y; }
  };
```

Class and Objects

- point.cc, point.cpp

```
void Point::setX(const int val) {
    _x = val;
  }

  void Point::setY(const int val) {
    _y = val;
  }
```

Main program

- main.cc, main.cpp

```
int main(int argc, char* argv[]) {
   Point apoint;

  apoint.setX(1);     // Initialization
  apoint.setY(1);

  // x is needed from here, hence, we define it here and
  // initialize it to the x-coordinate of apoint

  int x = apoint.getX();
}
```

Constructor and Destructor

```
class Point {
 private :
   int _x, _y;
 public:
   Point() {
     _x = _y = 0;
   }
   Point(const int x, const int y);
   Point(const Point &from);
   ~Point() {}

   void setX(const int val);
   void setY(const int val);
   int getX() { return _x; }
   int getY() { return _y; }
 };
```

Constructor and Destructor

```
Point::Point(const int x, const int y)  : _x(x), _y(y) {
}

Point::Point(const Point &from) {
      _x = from._x;
      _y = from._y;
}

Point::~Point(void) {
    /* nothing to do */
}
```

Standard Input/Output Streams

- Stream is a sequence of characters
- Working with cin and cout
- Streams convert internal representations to character streams
- >> input operator (extractor)
- << output operator (inserter)

Reading Data >>

- Leading white space skipped
- Newline character <nwln> also skipped
- Until first character is located

```
cin >> ch;
```

- Also read character plus white space as a character

CountChars.cpp

Program Output

```
Enter a line or press CTRL-Z: This is the first line.
This is the first line.
Number of characters in line 1 is 23
Enter a line or press CTRL-Z: This is the second line.
This is the second line.
Number of characters in line 2 is 24
Enter a line or press CTRL-Z: <CTRL-Z>
Number of lines processed is 2
Total number of characters is 47
```

CountChars.cpp (Header)

```
// File: CountChars.cpp
// Counts the number of characters and lines in
// a file

#include <iostream>
#include <string>

using namespace std;

#define ENDFILE "CTRL-Z”     //ENDFILE is a string
```

CountChars.cpp (Setup)

```
int main()
{
   const char NWLN = '\n'; // newline character

   char next;
   int charCount;
   int totalChars;
   int lineCount;

   lineCount = 0;
   totalChars = 0;

   cout << "Enter a line or press "
        << ENDFILE << " :"
```

CountChars.cpp (Main Loop)

```
while (cin.get(next)) {   // a new line, if user hits ^Z,
                          // cin.get returns 0
      charCount = 0;
      while (next != NWLN  &&  !cin.eof()){
    cout.put(next);
    charCount++;
    totalChars++;
    cin.get(next);
      }  // end inner while to get a line
  cout.put(NWLN);
  lineCount++;
  cout << "Number of characters in line "
       << lineCount << " is " << charCount << endl;
  cout << "Enter a line or press " << ENDFILE << ": ";
} // end outer while
```

CountChars.cpp (Output)

```
    cout << endl << endl
       << "Number of lines processed is "
       << lineCount << endl;
    cout << "Total number of characters is "
       << totalChars << endl;
  return 0;
}
```

File I/O

- Declare the stream to be processed:

```
#include <fstream>
ifstream ins;      // input stream
ofstream outs;     // output stream
```

- Need to open the files

```
ins.open(inFile);
outs.open(outFile);
```

Files

- `#define` associates the name of the stream with the actual file name
- `fail()` function - returns nonzero if file fails to open
- Program CopyFile.cpp demonstrates the use of the other `fstream` functions
  - `get`, `put`, `close` and `eof`
  - Copy from one file to another

CopyFile.cpp

Program Output

```
Input file copied to output file.
37 lines copied.
```

CopyFile.cpp (Header)

```
// File: CopyFile.cpp
// Copies file InData.txt to file OutData.txt

#include <cstdlib>
#include <fstream>

using namespace std;

// Associate stream objects with external file
// names
#define inFile "InData.txt"
#define outFile "OutData.txt"
```

CopyFile.cpp (Declarations)

```
// Functions used ...
// Copies one line of text
int copyLine(ifstream&, ofstream&);

int main()
{
   // Local data ...
   int lineCount;
   ifstream ins;
   ofstream outs;
```

CopyFile.cpp (Opening Input File)

```
// Open input and output file, exit on any
// error.
ins.open(inFile);
if (ins.fail ())
{
   cerr << "*** ERROR: Cannot open " <<
                  inFile << " for input." << endl;
   return EXIT_FAILURE;         // failure return
}  // end if
```

CopyFile.cpp (Opening Output File)

```
outs.open(outFile);
if (outs.fail()) {
    cerr << "*** ERROR: Cannot open " << outFile
        << " for output." << endl;
    return EXIT_FAILURE;        // failure return
}  // end if
```

CopyFile.cpp (Copy Line by Line)

```
   // Copy each character from inData to outData.
   lineCount = 0;
   do{
      if (copyLine(ins, outs) != 0)
         lineCount++;
   } while (!ins.eof());
   // Display a message on the screen.
   cout << "Input file copied to output file."
        << endl;
   cout << lineCount << " lines copied." << endl;
   ins.close();
   outs.close();
   return 0;        // successful return
}
```

CopyFile.cpp (copyLine procedure)

```
// Copy one line of text from one file to another
// Pre: ins is opened for input and outs for
// output.
// Post:    Next line of ins is written to outs.
//          The last character processed from
//          ins is <nwln>;
//          the last character written to outs
//          is <nwln>.
// Returns: The number of characters copied.
```

CopyFile.cpp (Character Reading)

```
int copyLine (ifstream& ins, ofstream& outs){
  // Local data ...
  const char NWLN = '\n';
  char nextCh;
  int charCount = 0;

  // Copy all data characters from stream ins to
  // stream outs.
  ins.get(nextCh);
  while ((nextCh != NWLN) && !ins.eof()){
     outs.put(nextCh);
     charCount++;
     ins.get (nextCh);
  }  // end while
```

CopyFile.cpp (Detection of EOF)

```
   // If last character read was NWLN write it
   // to outs.
   if (!ins.eof())
   {
      outs.put(NWLN);
      charCount++;
   }
   return charCount;
} // end copyLine
```