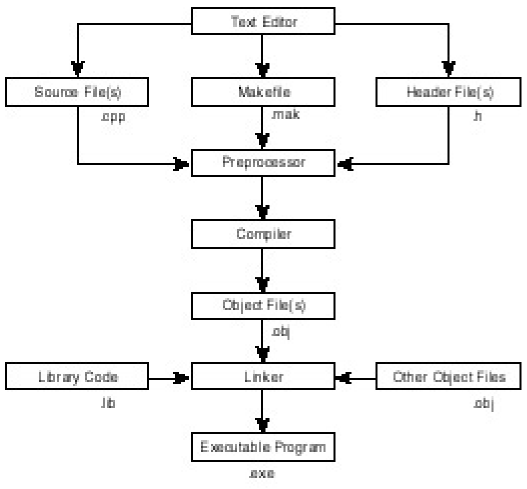
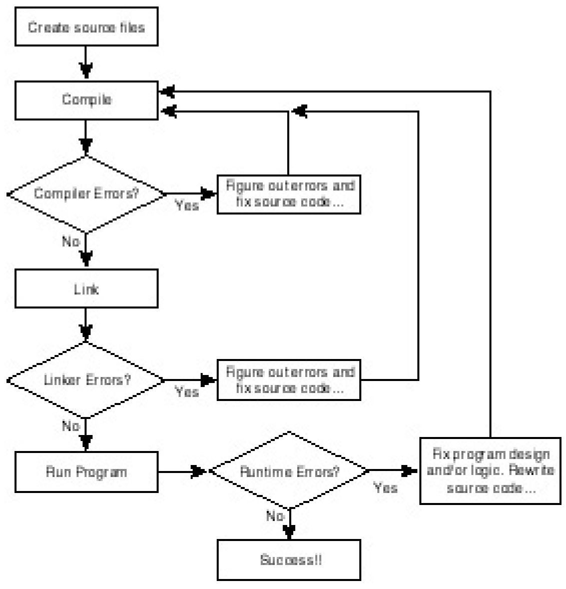
References

1. "Using C++" – Rob McGregor; QUE
2. "C++ for Programmers" – Leendert Ammeraal
3. "Special Edition Using Borland C++ 5" – Paul Kimmel; QUE
4. "C/C++ Languages and C++ Libraries" – Microsoft Visual C++ 6 Documentation
5. "Borland C++ 5 User's Guide" – Borland Corporation

Part I General C++ Programming

C++ Fundamentals




Create source files
Compile
Compiler Errors?
Yes
Figure out errors and fix source code...
No
Link
Linker Errors?
Yes
Figure out errors and fix source code...
No
Run Program
Runtime Errors?
Yes
Fix program design and/or logic. Rewrite source code...
No
Success!!

- *.h - Interface files (header files, include files)
- *.cpp - Implementation Files (source files)
- Process of building executables: (see diagram)
- Compiling: source code -> binary object files (*.obj)
- Linking: links the various compiled object files together into executables
- *.mak - make files: contains information about source files, obj files and libraries
- Overall Process of development: (see diagram)
- C++ is case sensitive
- Statements are followed by ;
- Function Prototype:
  `ret_type Func_Name ([Params]) BLOCK`
- A `BLOCK` is enclosed in curly braces {}
- `main()` – entry point of program execution
- Comments:
  `/* Comment */` - C-Style (multi-line, but cannot be nested)
  `// Comment` – C++-Style
- `iostream.h` – Input/Output Stream Lib
- `cin` object – read from keyboard (stdin)
  >> (extraction operator)
  eg. `cin >> name;  // name is the name of a variable`
- `cout` object – output to standard output (to screen by default)
  << (insertion operator)
  eg. `cout << "Welcome to C++";`
- `cerr` object – output error messages to screen
- Preprocessor directives – instructions to the preprocessor on how data is to be prepared for processing
- `#include` – one of the preprocessor directives that gives C++ access to libraries
  eg. `#include <iostream.h>`
- Escape Sequences: special characters prefixed by \ which carry specific function
  eg. `\n, \t, \a` etc.
- A hexadecimal number is preceded by 0x while an octal number is preceded by 0, eg. 0xFFFFFF; 0366
- C++ uses \ as a line continuation character

Variables & Constants

- C++ is strongly typed; type declaration is mandatory
- 2 categories of data types: Fundamental and Derived
- To declare a variable (and optionally assign a value to it, initialization):
  `datType varname [= value];`
- Bit – the smallest quantity; 1 Byte = 8 bits
- Variables declared within a block are *local* to that block, and are invisible outside the block. The lifetime of these variables extends into inner blocks, unless another variable of the same name is declared inside that block. As long as the called block terminates, the variables are back in scope.
- Variables declared inside functions are local to that function, while those declared outside any function (including main()) are global.
- Integer type: <NOTE> NOT to be delimited by commas
  Signed Integers (**Z**), Unsigned Integers (**N**)
- Integral data types (not involving floats): `bool, char, short, int, long`

TABLE 2.2 The Possible Ranges of Values for Signed and Unsigned Integral Data Types of Various Byte Sizes

Size	**Sign**	**Minimum Value**	**Maximum Value**
1	signed	–128	127
1	unsigned	0	255
2	signed	–32,768	32,767
2	unsigned	0	65,535
4	signed	–2,147,483,648	2,147,483,647
4	unsigned	0	4,294,967,295

- The `sizeof(datType)` function returns the size in bytes for a particular datType
- Use the `unsigned` type modifier to declare numerical variables that are always positive, eg:
  `unsigned [type] var;`
  < unsigned suffix: u >
- Character Data Type (`char`): represent a single character
  eg. `char ch = 'A';  // an equivalent way is to specify char ch = 65;`
- Character Sets – a set of characters used, vary from locale to locale
- A string is internally stored as an array of characters, with the size that is 1 more than the size of character string actually contained (owing to trailing null-terminated character). Therefore, for example, the word "GUITAR" can be equivalently expressed as
  ```
  char ch1[] = "GUITAR";
  char ch1[] = {'G', 'U', 'I', 'T', 'A', 'R', '\0'};
  char ch1[] = {0x47, 0x55, 0x49, 0x54, 0x41, 0x52, 0x00};
  ```
  If you type `cout << ch1 << "\n";` the output will all be GUITAR.
- The `string` class presents a better representation of strings, remember to include `#include <string>;`
- Boolean Data Type (`bool`): the two possible values are true or false
- Floating point & Double precision data types: `float, double, long double`, should affix "f" for `float`, "l" for `long double`; should include at least 1 decimal digit
- Examples:
  ```
  float MyFloat = 42.0F;
  long double MyLongDouble = 42.0L;
  double MyDouble = 42.0;
  ```
- Doubles are more mathematically accurate, but occupy larger memory size
- Derived data types are constructed from the fundamental data types, including arrays, pointers, references, structures, unions and classes.
- References: An alias of an object, mainly used to pass an object by reference
- Pointer: a variable that points to a specific location (address) in memory
- Structure: equivalent to a User-Defined Type (UDT) in VB, encapsulating its members

- Union: similar to structure, but only 1 member of the union can be accessed each time
- Literals are data elements that cannot be changed, e.g. 3, "A" etc.
- Use const function to declare and initialize constants; Uppercase constant nomenclature tradition; eg. `const double PI = 3.14159;`
- A character constant contain only 1 character or escape character, delimit with ' ';
- A String Literal contains zero or more characters surrounded by double quotation marks (") that act as a constant for a null-terminated string.
- enum (similar to that in VB) assigns a different constant to different elements in a group eg. `Enum CardSuits {Hearts, Spades, Diamonds, Clubs}; CardSuits cs; cs = Hearts;` Important Note: CardSuits is NOT an integer, thus, an explicit typecast from CardSuits to integer is necessary in Integer context.
- Declaration specifiers – specifies the data type portion of a declaration; declarators – specifies the name of identifier
- Type definitions – An alias declared for an existing data type through the use of the `typedef` keyword, eg. `typedef unsigned long ULONG;`
- You can use the `static` modifier in variable declaration to stipulate that the variable be maintained after the function is destroyed. A static variable is local to the function in which it is declared, but cannot be accessed from outside this scope. When the function is called again, the value of the static variable will be restored. Example:

```
#include <iostream>
void Increment(bool flag) {
    static int counter;
    counter++;
    if (!flag) cout << "Increment() has been accessed "
    << counter << " times.\n";
}
void main() {
    const int Iterations = 10;
    for (int j=0; j<Iterations; j++) {
        if (j==Iterations-1) Increment(false); else Increment(true);
    }
}   // Output: Increment() has been accessed 10 times.
```

- Sometimes it may be deemed necessary to embed some functions in source code which were syntactically correct in C but not C++, use eg. `extern "C" void MyFunc(int);`
- Coercing a data element from one type to another involves the process of *typecasting*, which can be done either explicitly or implicitly.
- Implicit casting occurs automatically whenever a *promotion* of data type is involved, eg. `char` -> `short` (1 byte -> 2 bytes) by assigning a `char` to a `short` variable.
- You can also explicitly instruct a typecast action, for example

```
long MyLong = (long) MyShort;    // MyShort is of short data type
```

Alternatively, `long MyLong = long(MyShort);`

Expressions & Statements

- Expression – any combination of operands and operators that produces a value
- Operations occur in a sequence as is stipulated by the *operator precedence*.
- In a *postfix expression*, the *postfix operator* (eg. ++) is appended to the end of a *primary expression*, which includes literals, identifiers, `this` pointers etc.
- Arithmetic operators: +, -, *, /, %, - (unary negation)
- Comparison & Logical operators: <, <=, >=, >, ==, !=, &&, ||, !
- Conditional operator (?:): eg.

```
int j = x > 0 ? x : y;  // j=x if x>0 and j=y if otherwise
```

- A Compound statement encloses a number of expression statements inside a block.
- Selection statements provide different pathways of execution in respond to the results of conditional statements.
- Selection statements can be nested.
- Iteration (Loop) statements can cause a code block to be executed a number of times.

- Jump statements transfer control away from the current point of execution.
- A label statement is actually an identifier that can be used to refer to an execution point in code, eg. LABELNAME:
- Expressions in C++ can evaluate to "l-values" or "r-values". L-values can be placed on the left-hand side of an assignment operator. For example, in `int k = 7`, `k` is an l-value. Named constants (that are declared with `const`) are l-values that cannot be modified.

Conditional Expressions & Selection Statements

- Evaluation of conditional statements result in a dichotomy that yields either true or false
- Boolean expression is enclosed in parentheses, eg.
  `bool MyBool = (false);` Alternatively, `bool MyBool = (0);`
- A zero (0) value is defined as `false`, and any non-zero values are evaluated to `true`.
- Relational Operators: == (equality operator), <=, >=, >, != (inequality operator)
- Logical Operators: && (logical AND), || (logical OR), ! (logical NOT, unary negation)
- Bitwise operators: &, |, ^, ~, << (shift left), >> (shift right)
- Selection statements are constructed through the use of `if` or `switch` statements.
- General syntax for `if`:
  ```
  if (expression-1) block_1
  [else if (expression-2) block_2] .. ..
  [else if (expression-n-1) block_n-1]
  [else block_n]      // If block contains only 1 statement, curly braces are
  optional
  ```
- General syntax for `switch`:
  ```
  switch (int_expression) {
  case constant-exp-1: {block_1; break;}
  case constant-exp-2: {block_2; break;}
  .. ..
  case constant-exp-n: {block_n; break;}
  [default: {block_n+1;}]
  }   // The default case is similar to #Case Else# in VB
  ```
  `int_expression` and `constant-exp-n` should be expressions that can be expressed in integral form (a char variable is also applicable).

Iteration Loops

- General syntax for `for`:
  ```
  for ([initializing-condition]; [loop_expression]; [altering-condition])
  block
  ```
  Looping will stop whenever the `loop_expression` returns false.
- General syntax for `while`:
  ```
  while (loop_expression) block
  ```
- General syntax for `do … while`:
  ```
  do block while (loop_expression);
  ```
- Use the `break;` statement to exit a loop regardless of if `loop_expression` is false.
- Use the `continue;` statement to head for the next loop, thus skipping the remaining loop statements.

Arrays

- An array contains a list of elements of the same data type, with the index starting from zero.
- To define an array, use the syntax `data-type arrayname[index-count]`,
  where `index-count` is the number of elements
- To access a particular element, use the syntax `arrayname[index]`
- To initialize a one-dimension array during declaration, eg. you can write
  `float num[] = {1.1f, 1.2f, 1.3f};`

- To get the number of elements in a char array (excluding null character) you can use the `int(strlen())` function as is defined in string.h
- In general, to create a multi-dimension array, use the syntax
  ```
  data-type arrayname[Dimension-1-count]…[Dimension-n-count];
  ```
- Example:
  ```
  const int MAXCOL = 3;
  const int MAXROW = 3;
  int Array2D[MAXCOL][MAXROW] = {
      {10, 20, 30},       // Column 1
      {40, 50, 60},       // Column 2
      {70, 80, 90}    // Column 3
  };
  ```
- To deduce the size of a <u>string</u>, use the `size()` or `length()` property.

Functions

- Functions – the art of abstraction
- The framework of a function:
  ```
  return-type-specifier Function-Declarator ([const] param-type-specifier
  param-declarator[ = default-value], …) Block
  ```
  The `const` modifier prevents the parameter from being changed inside the function.
- The `inline` modifier can be placed before function declaration to instruct the compiler to substitute the function call by the actual code that is executed, thus not suitable for lengthy functions. An inline function is generally processed faster.
- In a function whose return type is not `void`, use the `return` statement to return a value to the function caller.
- The `main()` function can receive command line arguments, in particular, it can accept two parameters, as below:
  ```
  void main(int argc, char *argv[]) Block
  ```
  `argv[0]` contains the complete path of the executable file. If there are additional parameters, `argv` will contain additional elements. `argc` contains the number of parameters, thus `argc` >= 1.
- The `exit(status)` function terminates the program with a value returned to the OS. `status` = 0 implies normal termination and abnormal termination if otherwise. Include `<stdlib.h>` header.

Function Overloading

- Function Overloading – establishing several functions sharing the same function name yet different sets of arguments, and the function that will eventually be invoked by a function call depends on the types and number of parameters that are sent to the function.
- Example: If we are to devise a function accepting two numbers as parameters which multiplies them together, we may write several overloaded functions for multiplication of different data types, eg. int-int; double-double; int-double etc.
- Overloading by different parameter types can be achieved, provided that the types of parameters, number of parameters or the return types of these overloaded functions are different.
- Technique: to round a float -> int correctly, you can write `int(num + 0.5f);`

Structures & Unions

- Structure is analogous to a User-Defined Type (UDT) in VB. Syntax for a union:
  ```
  struct StructTag {
      datType1 data-member_1;            A data member
      :       :   :
  datTypen data-member_n;
  };
  ```
- In C++, structures are actually classes with public access.
- Before populating a structure, you are required to first create an instance of the structure in the

form of a variable:
`StructTag StructVar;`

- To refer to a data member, use the syntax
  `structvar.datMemberId [=value];`
- You can populate a structure by assigning values to each individual data member. For example, for the structure
  ```
  struct Point3d {
      float x;
      float y;
      float z;
  } pt;
  ```
  you can write `pt.x = 1.0f; pt.y = 2.2f; pt.z = 3.0f;`
- We can as well create an array of structures, as in `pt[1000]`
- We can even include functions as members of a structure, for example
  ```
  struct Rectangle {
      int Left; int Right; int Top; int Bottom;
      int width() {   // This is analogous to a "method" in VB !!
          return (Right - Left);
      }
  } rect; // You can refer to the method by rect.width();
  ```
- Unions are similar in architecture to structures except that only one data member is assigned a value at any time. They have limited functionality yet are more efficient.
- For example, for a union variable `mu` containing the data members `num` and `flt`, if
  ```
  mu.num = 5000;
  mu.flt = 123.456f;
  ```
  After executing the above two statements, the values of both data members will be 123.456f. The original value of `mu.num` is thus lost.
- Nested structures and unions:
  ```
  union MyUnion {
      struct A {
          datType x;
          datType y;
      } var1;
      struct B {
          // internals omitted here …
      } var2;
  } mu;
  ```
  You can refer to x by `mu.var1.x [= value]`.

Pointers & References

- Pointers – a variable that stores a memory address.
- To declare a pointer, use `datType *pVariable;`
- To assign a variable's memory address to the pointer, prefix the variable by `&`, eg:
  ```
  int Var = 50;
  int *pVar;
  pVar = &Var;// The type must match !!
  ```
  Printing the value of `pVar` confirms that `pVar` is a hexadecimal number.
- To get the data out of a pointer, use the * to dereference it:
  eg. `cout << *pVar << endl;`
- Usage of the Pointer-to-member operator (->):
  ```
  Image Picture1;
  Image *pImage = &Picture1;
  Picture1.width = 100;      ┐
  (&Picture1) -> width = 100 │
  pImage -> width = 100;     ├ // These forms are equivalent !!
  (*pImage).width = 100;     ┘
  ```

- Passing a structure to a function Byval is inefficient, so in practice we use pointers instead.
  For example:
```
struct MyStruct {
    long bgcolor;
    long width;
    int cellpadding;
    int cellspacing;
};

void SetProps(MyStruct *ptag) {
    ptag->bgcolor = 0xFFFFFF;
}

void main() {
    MyStruct table;
    SetProps(&table);
}
```
- This method is also C-compatible.
- A reference is an alias for an object. Anything you perform on a reference variable is actually performed on the object itself.
- You must associate an object with the reference when you declare it. After this declaration the target of the reference can no longer be changed.
- Declare a reference: `datType &reference-variable = varname;`
- For example: if `float f = 2.2f; float &fr = f;`
  `fr *= 5` implies `f *= 5` yielding f = 11
- Passing an argument by reference to a function is as well supported in C++, to do so, affix the & operator before the variable name in the function declaration. Thus the previous example can also be achieved in this way:
```
struct MyStruct {
    long bgcolor;
    long width;
    int cellpadding;
    int cellspacing;
};

void SetProps(MyStruct &Tag) {
    Tag.bgcolor = 0xFFFFFF;
}

void main() {
    MyStruct table;
    SetProps(table);
}
```
- For parameters passing by reference that you wouldn't like to be modified, affix the `const` modifier. If you try to change them the compiler will return an error.
- A return value of a function can also be a reference. For example, in the program:
```
#include <iostream.h>

int &smaller(int &x, int &y) {
    return (x < y ? x: y);
}

void main() {
    int a = 23, b = 15;
    cout << "a = " << a << " ; b = " << b << endl;
    int &s_num = smaller(a, b);
    cout << "The smaller number is " << s_num << endl;
    s_num = 0;  // Note that you can write smaller(a,b) = 0 here
```

```
        cout << "The smaller of a and b is now set to 0.\n"
        "a = " << a << " ; b = " << b << endl;
    }
```

- When using this technique, the variable to be returned by reference should exist after the function call, thus returning local variables declared inside the function in this regard is not acceptable.
- An array, internally, occupies contiguous units of memory, according to the type size of the data stored. If we write `cout << s;` where s is an array, the starting byte offset (i.e. `&s[0]`) will be displayed. Since `s ≡ &s[0]` and `s + i ≡ &s[i]`, `*s ≡ s[0]` and `*(s + i) ≡ s[i]`. `s + i` means that i integers from the start offset, but not i bytes from offset, since the type size of data stored can vary.
- If we have `int j; char *p_char;` and try to write `p_char = &j;` it is incorrect because the type doesn't match. We need to do a cast: `p_char = (char *)&j;`
- In C++, we can establish a generic pointer (without type specification) by declaring the pointer with the void type modifier.

Dynamic Memory Allocation

- Static memory (stack) is a region that has been reserved by the compiler at compile time.
- Dynamic memory (heap) - memory available for use creating dynamic objects
- Heap-based objects have no predetermined life span, and will not be destroyed until you destroy them. If you do not destroy these objects before the program terminates, those memory occupied will be lost and result in a dreadful phenomenon of "memory leak".
- `new` operator – memory allocation; delete operator – memory de-allocation
- To create a dynamic object, use the syntax `objType *pVar = new obj;`
  For example, `Employee *emp = new Employee; // Employee is a structure`
  `char *pChar = new char('X');`
- The `new` operator returns the address of the newly created object in a pointer.
- To de-allocate memory occupied by a dynamic object, use the syntax `delete objPointer`. Ensure that the pointers are set to 0 afterwards. Also ensure that a delete operation is not carried out more than once, otherwise a system fault will be committed. For example, `delete pChar;`
  `pChar = 0;`
- A pseudo-dynamic array can be established like this:
  ```
  int count = 100;// hey! count is a variable !!
  char *str = new char[count];
  ```
  To trash it, use `delete [] str;`
- For multi-dimensional arrays, only the first dimension can vary, and higher dimensions have to be specified as a constant expression, as always. The following statements allocate a two-dimensional array with the size being NumFloats * 100:
  ```
  int NumFloat = 50;
  char (*pFloat)[100] = new float[NumFloat][100];
  ```
- In conclusion, static memory allocation stipulates that the size of data be known during compile time; while dynamic memory allocation allows the memory size to be adjusted subject to different necessities on different occasions (to be discussed later).
- Sometimes, the OS does not have, thus cannot allocate enough memory under your request, the program should be able to handle such a situation.
- A zero value in a pointer signals that insufficient memory is available.
- All pointers should be initialized with the value 0, especially if the pointers are not immediately assigned a valid memory address upon creation. An uninitialized pointer is undefined and accessing such a pointer may jeopardize the stability of the system.

Preprocessor

- Preprocessor – a program that is invoked by the compiler to process a source file.
- Preprocessing involves four phases, namely:
  1. Character mapping – converts any special character sequences into internal representation
  2. Line Splicing – maps contiguous lines, delimited with line continuation (\) characters, into a single

line
3. Tokenization – Analyses the source code and removes all program comments
4. Preprocessing – executes any preprocessor directives and then expands macros

- Preprocessor directives make it possible to write a single set of source files that can be compiled correctly on several operating systems and hardware platforms.
- Preprocessor directives within a macro are ignored.
- A list of preprocessor directives:

TABLE 12.1 **The C++ Preprocessor Directives**

Directive	Description
`#include`	Causes the preprocessor to include the contents of a specified file as if those contents had been typed into the source file in place of the directive
`#define`	Defines a preprocessor symbol in a program. A preprocessor symbol is a special identifier that the preprocessor can recognize.
`#undef`	Undefines an existing preprocessor symbol from your program
`#if`	Use with the defined() operator. Corresponds to the C++ if statement, but must be terminated by a matching #endif directive.
`#ifdef`	A Boolean directive that evaluates to true if a specified preprocessor symbol has already been defined.
`#ifndef`	A Boolean directive that evaluates to true if a specified preprocessor symbol has not yet been defined.
`#elif`	Used with the defined() operator, and in combination with the #if directive, this directive corresponds to the C++ else-if clause in an if statement.
`#else`	Used in combination with the #if directive. This directive corresponds to the C++ else clause in an if statement.
`#endif`	Terminates a corresponding #if directive.
`#error`	Generates a compile-time error message.
`#line`	Tells the preprocessor to change the line number and filename used internally by the compiler to some other specified line number and filename.
`#pragma`	A directive that specifies compiler-specific instructions to the compiler, allowing it to offer machine-specific and operating system-specific features, yet still retains compatibility with C or C++.

- Directives can be placed anywhere in a source file, yet a directive can only exert its effect to statements appeared after the directive.
- `#define` can be used for text substitution macros, as in
```
#define PI 3.14159
#define HKU "The University of Hong Kong"
```
They can then be used just like any other constants.
- Constants defined like this do not require a type specified, and may lead to subtle coding errors.
- You can use `#ifdef` or `#ifndef` to check if a symbol has been previously defined:
```
#ifndef __BYTE
#define __BYTE
typedef unsigned char __BYTE;
#endif
```
- Another use is for conditional compilation of different portions of code for different compilers:
```
#ifdef __BORLANDC__     // Borland C++
#include <mem.h>
#elif defined(_MSC_VER) // Microsoft C++
#include <memory.h>
#endif
```
- Apart from writing `#ifndef`, you may write like this:
```
#if !defined(SYMBOL)
// do something …
#endif
```
- The technique of Single-File Inclusion (prevent multiple file inclusion):
borrowing the internal method employed in *name mangling*:
1) Create a header file. Then create an identifier with the nomenclature following this pattern:
__FILENAME_EXT__

eg. __MYSTRUCT_H__

2) Put the following statements in the header:

```
#ifndef __FILENAME_EXT__
#define __FILENAME_EXT__
```

3) Include all definitions, and close by adding #endif. The following shows a finished header example:

```
// MYSTRUCT.H
#ifndef __MYSTRUCT_H__
#define __MYSTRUCT_H__
typedef struct MyStruct {
int member1;
float member2;
}
#endif  // __MYSTRUCT_H__
```

- See references for specific compilers for a list of predefined macros.
- `#pragma` is more often used by advanced programmers and will not be discussed here.
- # operator can convert pre-expanded macros into a string by adding the double quotes. For example, if you define
  ```
  #define PRINT(a) cout << #a << '\n'
  ```
  in a header, in a source file you can write
  ```
  PRINT(  A few words here  ); // The spaces are ignored
  // OUTPUT: A few words here
  PRINT("Quoted words");       // OUTPUT: "Quoted words"
  ```
- ## operator (concatenation operator) can paste two tokens together, eg.
  ```
  #define DEF(a)           int Test##a
  #define PRINT(a)     cout << Test##a << '\n'
  ```
  Writing `DEF(1) = 5;` is equivalent to writing `int Test1 = 5;`
  `PRINT(1);` is equivalent to writing `cout << Test1 << '\n';`
- There are six predefined macros:

TABLE 12.5 **The six ANSI C Predefined Macros**	
__DATE__	The compilation date of the current source file
__FILE__	The name of the current source file
__LINE__	The line number in the current source file
__STDC__	Specifies complete conformance to ANSI Standard C specification. Not defined for C++ compilers.
__TIME__	The most recent compilation time of the current source file
__TIMESTAMP__	The date and time of the last modification of the current source file

Function Pointers

- Functions defined in a program will be loaded into the memory at runtime; therefore, every function yields a *function address*, which marks the beginning of the function in the memory.
- To declare a pointer to a function you use the syntax:
  ReturnType (*Name) (ParameterList)
  eg. `void (*pFunc) (int i);`
  `// for a function whose prototype is void Func(int i);`
- Now you can assign the function address to the pointer:
  `pFunc = Func;`
- To invoke the function with the pointer, use either syntax:
  eg. `(*pFunc) (5);`   OR   `pFunc(5);`
  If you use the second syntax, the pointer will be dereferenced implicitly.
- If you send this function pointer to a Test() function as an argument, it looks like this:
  `void Test(void (*pFunc) (int i));`
  which is very cumbersome. To simplify the expression, use the `typedef` keyword:

```
typedef void (*PFUNC) (int i);
```

Thereafter you can write `void Test(PFUNC pFunc);` instead.

- Function Pointers can reduce code repetition by allowing you to make tables of functions to call based on runtime conditions (eg. Using a `switch()` statement).
- An array of function pointers can also be established.
- How to set up pointers to overloaded functions?
  For overloaded functions
  ```
  int Overload(int j, float f);
  double Overload(double d, float f);
  ```
  You can set up different typedefs
  ```
  typedef int (*POVERLOAD1) (int j, float f);
  typedef double (*POVERLOAD2) (double d, float f);
  ```
  and then setting up pointers like these:
  ```
  POVERLOAD1 pfn1 = Overload;
  POVERLOAD2 pfn2 = Overload;
  ```
  You can then call the overloaded function via pointers:
  ```
  pfn1(3, 3.0f);
  pfn2(4.0, 3.0f);
  ```
- The following example shows the use of pointers to member functions:
  ```
  #include <iostream>

  // The Box structure
  struct Box {
      float length;
      float width;
      float height;
      float CalcVolume() {
          return length * width * height;
      }
  };

  // Function pointer types
  float (Box::*PFN1)() = &Box::CalcVolume;
  typedef float (Box::*PFN2)();

  void main() {
      Box cube = {2.0f, 2.0f, 2.0f};
      // Declare and initialize function pointer (from typedef)
      PFN2 pfn = &Box::CalcVolume;

      // Call the function via pointer using different approaches
      cout << "\nPFN1: Cube volume is " << (cube.*PFN1)();
      cout << "\nPFN2: Cube volume is " << (cube.*pfn)();
  }
  ```

Operator Overloading

- In C++, we can give a new definition to an existing operator specific to our user-defined types (or classes). This is called operator overloading. For example, the operators << and >> of the `cout` and `cin` objects, respectively, are actually overloaded operators.
- We overload an existing C++ operator, in which a function will be executed when a specified operator is encountered in an expression.
- The general syntax for creating (overloading) an existing operator is:
  ```
  ReturnType operator OpSymbol();
  ```
  eg. if I want to define a symbol for addition of vectors, I can declare
  ```
  vector operator+(vector &a, vector &b);
  // provided the vector type has been established
  ```
- Example 1:

```
struct Point2d {
    float x;
    float y;
};
// Function prototype
Point2d operator-(const Point2d &p1, const Point2d &p2);
// The overloaded function
Point2d operator-(const Point2d &p1, const Point2d &p2) {
    Point2d pt;
    pt.x = p1.x - p2.x;
    pt.y = p1.y - p2.y;
    return pt;
}
void main() {
    Point2d pt1 = {10.0f, 16.0f};
    Point2d pt2 = {8.0f, 7.0f};
    Point2d pt3 = pt1 - pt2;
    // output pt3 omitted here …
}
// pt3 = {2.0f, 9.0f};
```

- Example 2: (Classes will be discussed later)

```
#include <iostream>

class vector {
private:
    float xx, yy;
public:
    vector(float x=0, float y=0) {xx = x; yy = y;}
    void printvec();
    void getvec(float &x, float &y) {x = xx; y = yy;}
};

void vector::printvec() {
    cout << xx << ' ' << yy << endl;
}

vector operator+(vector &a, vector &b) {
    float xa, ya, xb, yb;
    a.getvec(xa, ya); b.getvec(xb, yb);
    return vector(xa + xb, ya + yb);
}

void main() {
    vector u(3, 1), v(1, 2), s;
    s = u + v;
    s.printvec();   // output 4  3
}
```

- If you are to overload an assignment operator, the overloaded function has to be a member function of the structure or the class. Operators that are class members can take only one argument.
- In operator overloading, it is important to consider whether an operator will be used as an l-value. For example, when we use an overloaded [] operator like this:
  `vector[9] = 2;`
  The operator function should return a reference type. Another example is the assignment operator. For example, this syntax is valid in C++:
  `a = b = c = 1;      // provided a, b, c has been declared as int already`
  You have to handle similar situations when you are overloading the assignment operator for custom types.

Namespaces

- When you use multiple external libraries, it is likely that there will be two identifiers sharing the same name. Namespace is a C++ declaration area that lets you add a secondary name to an identifier, thus eliminates the occurrence of possible name clashes.
- Syntax for declaring a namespace:
  `namespace name {memberlist}`
  for example, in two separate modules, we can declare like this:

```
// module1.h
namespace Physical {
    struct Pencil {
        float length;
        int color;
        bool eraser;
    }
}
```

```
// module2.h
namespace Retail {
    struct Pencil {
        float price;
        char *manufacturer;
        int id;
    }
}
```

- We can then use the scope resolution operator (::) to specify which structure to use:
  `Physical::Pencil` and `Retail::Pencil`
- A namespace is just a marker of certain area, so you can include any C++ statements inside a namespace, and you can split the namespace in different parts of code too.
- The `using` declaration allows you to access some member of a certain namespace without explicitly specifying the identifier of the namespace. (qualifying with explicit names), eg: `using Physical::length; //Thereafter you can specify length directly`
- If a global variable `length` is also defined, you can specify `::length`
- If several `using` statements are in nested blocks, the scope of effectiveness of these using statements follow the case of variable declaration, example:

```
#include <iostream>
// namespace 1
namespace first {
    void func1();
    void func1() {
        cout << “first::func1()” << ‘\n’;
    }
}

// namespace 2
namespace second {
    void func1();
    void func1() {
        cout << “second::func1()” << ‘\n’;
    }
}

// Global function
void func1();
void func1() { cout << “::func1()” << ‘\n’; }

void main() {
    using first::func1;
    func1();        // first::func1() will be executed
    {
        using second::func1;
        func1();    // second::func1() will be executed
        using ::func1;
        func1();    // ::func1() will be executed
    }
    func1();        // first::func1() will be executed
}
```

- The `using` directive, on the other hand, makes all identifiers in the specified namespace available. This is especially convenient if you have to use a large number of identifiers in that namespace. Syntax: `using namespace name;`
- After the `using namespace` statement, You can override the default by using :: again:
  Example:
  ```
  using namespace first;
  func1();        // This line calls first::func1()
  second::func1();// This line calls second::func1()
  ```
- In the case above, if a global func1() exists, there will be a name clash – use explicit qualification in this case.
- It would be difficult to refer to lengthy namespace identifiers. You can employ the preprocessor technique of using `#define ShorterName LongerName` to create an alias.
- We can nest namespaces. In this cascading namespace:
  ```
  namespace Level1 {
      namespace Level2 {
          namespace Level3 {
              void func1() { }
          }
          void func1() { }
      }
      void func1() { }
  }
  ```
  To access them, we need to write, respectively,
  ```
  Level1::func1();
  Level1::Level2::func1();
  Level1::Level2::Level3::func1();
  ```
- If we don't specify the namespace identifier (as in ::func1()), the *global namespace* is assumed.
- The Standard Namespace, with the identifier std, wraps the entire standard C++ library.

Manipulating Bits

- The leftmost bit is the most significant bit while the rightmost bit is the least significant bit
- A *word* comprises 16 bits, while a *double word* occupies 32 bits. A double word can be split into a high word (bit 16-31) and a low word (bit 0-15).
- A special type of structure data member contains a specified number of bits, called bit field
- To declare a bit field, use syntax `DataType Name : size;`
  where `DataType` is confined to all integral types. Example:
  ```
  struct MyBit {
      unsigned short b1 : 1;  // 1 bit allocated
  } mb;
  ```
  You can refer to b1 by `mb.b1`
- The bitwise operators are |, &, ~, ^ (XOR), <<, >>
- Applications of **&**:
  1) masking (filtering) specified bits from an operand;
  2) check if a number exceeds a particular limit.
  Examples:
  1) `0xDA8019 & 0x00000F` yields `0x000009`
  2) `((0xDA8019 & 0x00000F) == 0x00000F)` yields `false`
  It is obvious that if a number is < `0x00000F`, the test above yields `true`.
- Application of **|**:
  It is very common in computers to use a sequence of bits for storage. In UNIX you will see file access mode like this: `-rwxr-xr-x (755)`
  Why is it represented by 755? The only leftmost character permitted is `d` (directory). The next three denote read, write and execute access permissions by the owner. The next three are for group access. The rightmost three are for others. `r` carries the value of 4, `w` carries the value of 2 and `x`

carries the value of 1. `d` does not carry any value and is specified by the system. Thus, adding up the three values in each group and omitting the first character you will get 755. If you specify it in binary format you will get 111 101 101. Actually, this value is obtained by using | to combine the values of individual bits together, like this:

```
          100 000 000   }
          010 000 000   }  Owner
          001 000 000   }
          000 100 000   }
          000 001 000   }  Group
          000 000 100   }
|         000 000 001   }  Other
--------------------
          111 101 101
```

- The above technique can only be applied to a bit whose value is 1. We need to devise a reliable way to change a certain bit to 0, regardless of the original value of the bit concerned. That is, you need to ensure that,
  eg. to ensure a value 0 in bit 2 (start from 0 from the right):
  `1001 -> 1001` and `1101 -> 1001`
  SCHEME: `1001 & (1111 ^ 0100)` <w/o using ~>
  (reasoning: preserve bit 0, 1, 3 and put 0 to bit 2)
  ALTERNATIVE: `1001 & ~0100          // seems less complicated`
  Both schemes employ the principle that `X & 0 = 0` and `X & 1 = X`

OOP – Introduction

- Characteristics of an object are defined by its corresponding class.
- A runtime instance is an object variable created at runtime. During program execution, objects can then be created according to the class specification (definition).
- Encapsulation is used to describe the situation in which data are contained within the scope of an object.
- An object exposes itself to other objects through the class interface.
- The process of creating a new class from an existing class is called inheritance, in which properties and methods of the object are inherited.
- Inheritance provides the new, derived (child) class with all the functionality contained in the base (parent, or ancestor) class, and allows you to extend the class's capabilities to provide solutions specific to the new class.
- Polymorphism is the potential of different objects in a class hierarchy to respond differently to the same message.
- Inheritance is implemented through the use of virtual methods, which allow a derived class to change the implementation of the method by overriding the original behaviour of the ancestor class.
- Carelessness in defining classes easily results in the dreadful phenomenon of "code bloat", as the inheritance hierarchy gets more complicated. Therefore, an ancestor class should not define any members that won't be used in the descendents.
- OOP helps enforcing code reuse. This saves development effort and time.

OOP – Classes

- An access specifier defines the type of data access permitted to class members. It can be `private`, `public` or `protected`.
  `private`: class members are accessible by the class itself
  `public`: class members are accessible by any object
  `protected`: class members are accessible by the class itself and all its child classes
  Friend classes or friend functions can access all members inside a class.
- A class binds its members into a cohesive unit. It aids in abstraction by hiding the complexities from users.
- To declare a class, use the syntax:

```
class clsMyClass {
    // class definitions here …
};
```

- <See example 2 in "Operator Overloading">
- Classes can be nested, as in structures.
- The `static` modifier stipulates that all instances of the class share a single copy of the data member. Example:

  ```
  class clsPoint3d {
  public:
      float x; float y; float z;
  protected:  // can only be accessed by class and derived classes
      static int count;   // count the number of class instances running
  };
  ```

  To initialize the value of `count`, you can place this statement in file scope:

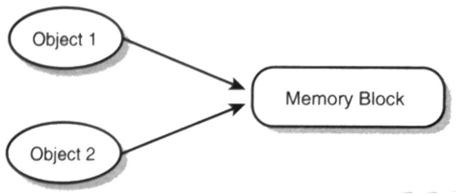
  ```
  int clsPoint3d::count = 0;
  ```

  File scope is defined as the region that is outside of all blocks or classes. However, after initialization you may change the value anywhere, including the class scope.
  Static members cannot be declared virtual.
- Constructor classes and destructor classes are member functions called <u>automatically</u> when an object is being created and before it is being destroyed, respectively. They have no return type specifiers.
- Constructor functions:
  - Default constructors: no parameters typically, default values required if otherwise
  - Have the same name as the class
  - The initializer list is a list of values used by a class constructor to initialize data member values before the body of the constructor executes.
  - Can be overloaded
  - Static members can only be initialized outside the class declaration.
- Destructor functions:
  - Nomenclature: ~ followed by class name
  - Purpose: clean up any dynamically allocated memory etc.
  - Cannot be overloaded
- The `this` pointer:
  - The `this` pointer is an internal pointer that yields the address of an object instance. It is used by member functions to identify a specific instance of a class.
  - Can only be used in non-static member functions
  - The `this` pointer points to the object instance from which the class member is invoked. Syntax: `this->ClassMember`
- Copy constructors:
  - A copy constructor is a constructor function designed to copy objects of the same class type.
  - accept a single argument (a reference to the same class type) and returns a copy of an object
  - The copy constructor is called whenever your code passes an object to a function by value instead of by reference. The copy constructor is also called if you initialize a new object with another object of the same type
- Example:

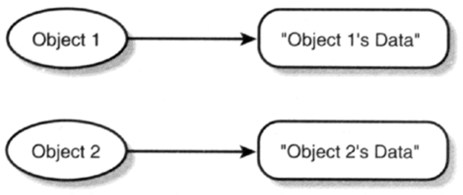
  ```
  #include <iostream>

  class MyClass    // class declaration
  {
  public:
    MyClass();    // default Constructor
    MyClass(const MyClass &myclass);    // Copy Constructor
    ~MyClass();    // destructor

    float myFloat;
    float *myFloatPtr;
  };
  ```

```
MyClass::MyClass() :// default Constructor (for pmc)
myFloat(0.0f) { // initializer list
  myFloatPtr = new float;
  *myFloatPtr = 0.0f;
}

MyClass::~MyClass() {   // Destructor
  if (myFloatPtr) {
       delete myFloatPtr;
       myFloatPtr = 0;
  }
}

MyClass::MyClass(const MyClass &mc) {    // Copy constructor
//  myFloat  = mc.myFloat;          // What will happen if you do this?
//  myFloatPtr = mc.myFloatPtr;  // ANSWER: Shallow Copy!

  myFloat  = mc.myFloat;
  myFloatPtr = new float;
  *myFloatPtr = *mc.myFloatPtr; // assign by VALUE
}

void main() {
  MyClass *pmc = new MyClass;    // dynamically create pmc
  *pmc->myFloatPtr = 10.0f;
  pmc->myFloat = 9.0f;

  MyClass mc(*pmc);    // now we use copy constructor
  cout << "The value of *myFloatPtr is: "
      << *mc.myFloatPtr << '\n';

  delete pmc;

  cout << "The value of *myFloatPtr is: "
      << *mc.myFloatPtr << '\n';
}
```

- There are two types of copying: shallow copying and deep copying.
- Shallow Copy (bitwise copy) means that the number and state of the data bits in one object are exactly reproduced in a second object. When pertaining to a pointer, a shallow copy causes the address stored in the pointer to be copied, and results in a situation where two pointers refer to the same block of memory. If any object pointer is destroyed data in the memory block will also be released. Since the remaining pointers are still non-zero, they are still pointing to an undefined block of memory. Calling delete on these pointers with the resource already deleted can cause havoc on system stability.



- In deep copy, on the other hand, you allocate new resources adequate enough to store a copy of the source object data and duplicate the data into the new memory location. As illustrated in the figure on the right, the data itself, rather than merely the address, is copied so that if one of the objects is destroyed, the remaining object still points to some valid data.



- We can overload an operator for use with a class by creating a member operator. Example:

```
CPoint3d &CPoint3d::operator=(const CPoint3d &pt) {
     x = pt.x; y = pt.y; z = pt.z;
     return *this;
}
```

- In the above example, if `CPoint3d` is a class containing `x, y, z` as members, we can write `CPoint3d pt2 = pt1;` to copy the contents of object pt1 to pt2.
- More Pointer-to-member Operators: `->*, .*`
  - Syntax:
    ```
    ClassObject .* ClassMemberPointer
    ClassObjectPointer ->* ClassMemberPointer
    ```
  - Examples:

```
#include <iostream.h>
const char NL = '\n';

class mycls {
    public:
        int member;
        int *ptr;
};

// Derive a pointer type to the non-pointer class member
int mycls::*pmember = &mycls::member;

void main() {
  mycls MyClass;  // Create a class instance

  // Allocate a ptr member
  MyClass.ptr = new int;

  // Get a pointer to the class instance
  mycls* pMyClass = &MyClass;

  // Give the members values
  pMyClass->*pmember = 5;
  *MyClass.ptr   = 10;

  // Get the member values
  cout << "pMyClass->*pmember = "     << pMyClass->*pmember << NL;
  cout << "MyClass.*pmember  = "  << MyClass.*pmember << NL;
  cout << "*MyClass.ptr   = "     << *MyClass.ptr << NL;
  cout << "*pMyClass->ptr  = " << *pMyClass->ptr << NL;

  // Clean up
  delete MyClass.ptr;
}
```

Note that `*pmember` in the above example can <u>only</u> accommodate `int` members of the class `mycls`.

OOP – Inheritance

- Example:

```
Class CBase {
    protected:
        int myInt;
};

// creates a new class derived from CBase
Class clsDerived : public CBase {
```

```
    protected:
        float myFloat;
};
```

- Constructors and destructors are called for each class in an inheritance hierarchy. Base classes would be the first to be created, and the last to be destroyed.
- You can *override* a function that is inherited from an ancestor class. This allows you to change the implementation of the function within the derived class. The overriding function has the same function signature as the overridden function (ancestor function). To specify which function is to be called in this case, use the scope resolution operator (::).
- A friend function is a function external to a class that has full access to all class members, including those that are protected and private. Since a friend function is external to a class, it will not be inherited into derived classes.
- A *friend class* is a class whose member functions are all friend functions of some class.
- By using a binary library of reusable classes (class library), large projects become more manageable. To use the classes, only the library file and necessary headers are required.

OOP – Polymorphism

- Polymorphism is the potential of different objects in a class hierarchy to respond differently to the same message.
- A virtual member function in a base class allows a derived class to change the implementation of the method. Descendants of the ancestor class can override the implementation of a base class's virtual function as long as the descendant uses the same function signature. Once a function is declared as virtual, you can redefine it in its derived class, even if the number and type of arguments are the same.
- The `virtual` modifier is to be placed in the class declaration only, eg.
  ```
  class cBase {
  protected:
      virtual void func1();
  };
  ```
  while in the function implementation, only `void cBase::func1() { // … }` is required.
- An *abstract base class* is a C++ class that contains at least one pure virtual function. An abstract base class is designed merely as a generic blueprint for more specialized classes, and is *not* meant to be used itself. No objects can be directly created from it. Derived classes should supply their own implementation of these virtual functions.
- A pure virtual function is a virtual function declaration in a base class that is set equal to zero. Example:
  ```
  virtual void Draw() = 0;
  ```
  It is up to derived classes to provide an implementation for the function.
- A derived object contains a lookup table called a virtual function table (also called a v-table, or VTBL). This table contains entries that point to the memory addresses of any virtual functions inherited from its ancestors. When an inherited function is called, the object looks into its v-table and uses the function pointer stored there to call the proper version of the function from its ancestor code.
- Examples:
  ```
  CBase *obj = new CBase;
  obj->func1(); // The CBase VPTR points to CBase:func1()
  CDerived *obj = new CDerived;
  obj->func1(); // The CDerived VPTR points to CDerived:func1()
  CDerived *obj = new CDerived;
  obj->CBase::func1(); // calls the inherited CBase::func1()
  ```
- A pointer to the base class can be pointed to a derived object.
  Example: `CBase *obj = new CDerived;`
  This feature facilitates the occurrence of polymorphism. For example, in the example below, `CBase2::Draw()` is declared with `virtual`, this instructs the runtime system to locate the correct version of `Draw()` to be executed:

```
/////////////////////////////////////////////////////////
```

```
//  Module  : virtual.cpp
//
//  Purpose : Demonstrates how a pointer to a base
//            class calls derived class functions for
//            virtual and nonvirtual base class
//            functions.
////////////////////////////////////////////////////////

#include <iostream>

////////////////////////////////////////////////////////
//  Class CBase1

class CBase1
{
public
   void Draw()
      { std::cout << "CBase1::Draw()\n"; }
};

////////////////////////////////////////////////////////
//  Class CDerived1

class CDerived1 : public CBase1 // Derive from CBase1
{
public:
   void Draw()
      { std::cout << "CDerived1::Draw()\n"; }
};

////////////////////////////////////////////////////////
//  Class CBase2

class CBase2  // Uses virtual Draw() function
{
public:
   virtual void Draw()
      { std::cout << "CBase2::Draw()\n"; }
};

////////////////////////////////////////////////////////
//  Class CDerived2

class CDerived2 : public CBase2 // Derive from CBase2
{
public:
   void Draw()
      { std::cout << "CDerived2::Draw()\n"; }
};

////////////////////////////////////////////////////////
//  main()

void main()
{
   // Create & use CBase1/CDerived1 object
   //
   CBase1    *base1;
   CDerived1  derived1;
```

```
    base1 = &derived1;

    std::cout << "base1->Draw() calls   : ";
    base1->Draw();
    std::cout << "derived1.Draw() calls : ";
    derived1.Draw();

    // Create & use CBase2/CDerived2 objects
    //
    CBase2    *base2;
    CDerived2  derived2;

    base2 = &derived2;

    std::cout << "\nbase2->Draw() calls   : ";
    base2->Draw();
    std::cout << "derived2.Draw() calls : ";
    derived2.Draw();
}
```

In this example, the output is:

```
base1->Draw() calls   : CBase1::Draw()
derived1.Draw() calls : CDerived1::Draw()

base2->Draw() calls   : CDerived2::Draw()
derived2.Draw() calls : CDerived2::Draw()
```

- Polymorphism only works with pointers and references. Passing polymorphic objects to functions by value will result in copy constructors being called. Moreover, consider the following example:
  ```
  // CCylinder is a derived class of CShape3d
  CShape3d *shape = new CCylinder;
  CShape3d shape2 = *shape;
  ```
  "class slicing" occurs in which a `shape2` object is assigned a `CShape3d` object, rather than `CCynlinder`, because the copy constructor defined in `CShape3d` does not know the extended definitions in `CCylinder`. To prevent this situation, always pass polymorphic objects by reference or pointer instead.
- If you polymorph a base class pointer into a derived class pointer, and the base class doesn't have a virtual destructor, the derived class destructor is never called. Declare a base class's destructor as virtual eliminates the problem of potential memory leak.

Dynamic Memory Allocation (cont'd)

- Container classes are C++ classes that act as containers for other objects, such as an array or a linked list.
- A list is a linear sequence of items.
- The earlier concept of creating a "dynamically-sized" array using the `new` keyword still does not allow us to create an array that can be resized dynamically so that only as much memory as required is allocated to the array.
- One of the solutions is to create a dynamic array class.
  - A dynamic array class is actually a class containing a pointer to a dynamic array and a property storing the number of elements inside the dynamic array, both protected members.
  - In the constructor, the dynamic array pointer and the size are to be initially assigned 0.
  - We have to write several methods for this class: Add(), Remove() …
- A *linked list* is a special collection structure that holds objects that are linked to other objects by pointers. The list is just a linear chain of objects called *nodes*. A *singly linked list* contains nodes that store some type of data, and each node has a pointer (the link) to the next node in the list. The first node in the list is called the *head*. The advantage of a linked list over an array is that while all array elements must be contiguous, the nodes in a list can reside anywhere in memory. Each node links to the next node using a pointer to that node. In a doubly linked list each node contains links (pointers)

to the previous node in the list and the next node in the list. A doubly linked list can be traversed both forward and backward. The advantage of a linked list over an array is that the elements are linked by pointers, and rearranging the order of the elements can be easily accomplished by updating the relevant pointers.

Templates

- The use of templates allows C++ programmers to write generic code that will work with multiple data types. In abstract terms, templates are *parameterized types.* A template uses the parameter or parameters sent to it to set up a type-specific version, called a *specialization* of the template, at compile time.
- Function overloading just eliminates the problem of devising multiple function identifiers, yet it does not help in reducing redundant code in general.
- To create a function template, use the syntax:
```
template <class T>
ReturnType FunctionName(ParametersList)
```
- Example:
```
template <class A, class B>
A Multiply(A num1, B num2) { return num1 * num2; }
```
- The above example shows how to construct a function template. The compiler generates a type-specific version of the template (called specialization) when it encounters an instantiation of a template object. In the example, as different sets of A and B are given, different sets of overloaded `Multiply()` function are established.
- Similarly, we can set up a class/struct template, e.g.
```
template <class T>
class BasicClass {
public:
   BasicClass(const T& t) : myObj(t) { }
   T Get() const { return myObj; }
   void Set(const T& t) { myObj = t; }
protected:
   T myObj;
};
```
- We can then instantiate a `BasicClass` object that is specialized to use a `float` parameter type like this:
```
BasicClass<float> f;
```
By providing a typedef for a specialized class template, the code can be made more readable and maintainable. Example:
```
typedef BasicClass<float> Float;        // be careful of capitalization
```

You can then use the specialized template typedef, like this:
```
Float f(5.5f);
cout << f.Get() << endl;

f.Set(12.3f);
cout << f.Get() << endl;
```
- Another use of a template is when we have to vary a particular quantity (we can accomplish this with an overloaded constructor, though). Example:
```
#include <iostream>
#include <math>

template <class T, int DIMENSIONS = 3>
class VECTOR {
    public:
    T& operator[] (int subscript) {
        return ((subscript < DIMENSIONS) ? data[subscript]:
        data[DIMENSIONS]);
    }
    protected:
    T data[DIMENSIONS + 1];
};

void main() {
    const int INT_DIM = 10;
    typedef VECTOR<int, INT_DIM> v10d;
    v10d vtemp;
    for (int i = 0; i < INT_DIM; i++) vtemp[i] = (int)pow(i + 1, 2);
    for (int i = 0; i < INT_DIM; i++)
        cout << "vtemp[" << i << "] = " << vtemp[i] << '\n';
}
```

- In the code snippet above, you put the class definitions inside the class. To put the `operator[]` function outside the class, for example, you may add:

```
template <class T, int DIMENSIONS = 3>
T& VECTOR<T, DIMENSIONS>::operator[] (int subscript) {
    return ((subscript < DIMENSIONS) ? data[subscript]: data[DIMENSIONS]);
}
```

  Do not leave out the "template" line.