C++

- A better C
- A superset of C
- Created at Bell Labs in the 1980's and called C with Classes
- Adds additional features to improve the language
- Adds functions and features to support Object Oriented Programming (OOP)

C v/s C++

- Excessive use of global variables (variables known throughout the entire program) may allow bugs to creep into a program by allowing unwanted side effects.
- *The concept of compartmentalization is greatly expanded by C++. Specifically, in C++, one part of your program may tightly control which other parts of your program are allowed access.*

C++

- The C++ standard library can be divided into two halves:
  - the standard function library

    &
  - the class library.

  The standard function library is inherited from the C language.

C++

- The C++ class library provides object-oriented routines that your programs may use.
- It also defines the Standard Template Library (STL), which offers off-the-shelf solutions to a variety of programming problems.

C++

- Traditionally, C programs use the file extension .C, and C++ programs use the extension .CPP.
- A C++ compiler uses the file extension to determine what type of program it is compiling.

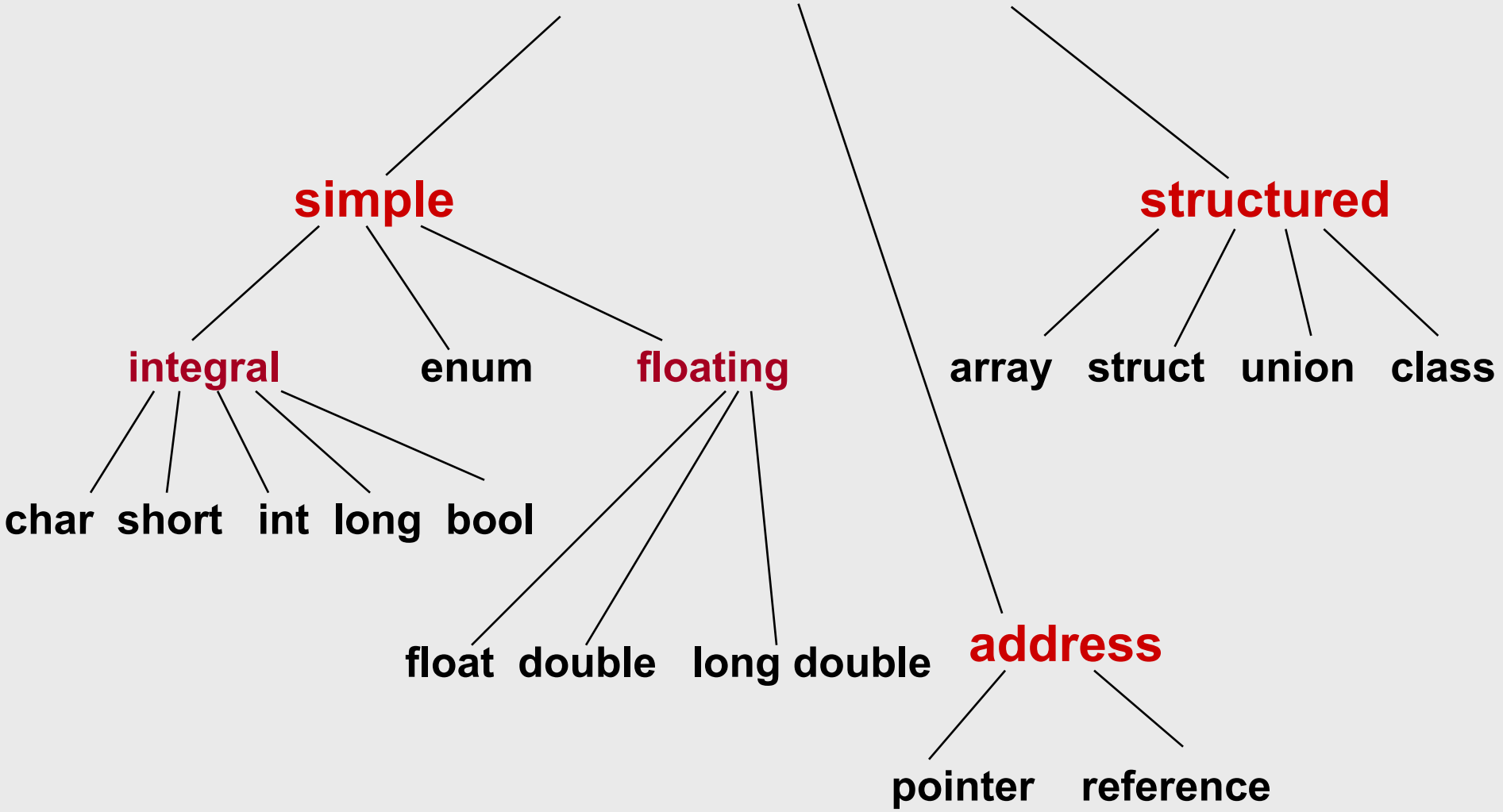
C++

- In C++, there is no limit to the length of an identifier, and at least the first 1,024 characters are significant.
- In an identifier, upper- and lowercase are treated as distinct. Hence, **count**, **Count**, and **COUNT** are three separate identifiers.
- An identifier cannot be the same as a C++ keyword, and should not have the same name as functions that are in the C++ library.

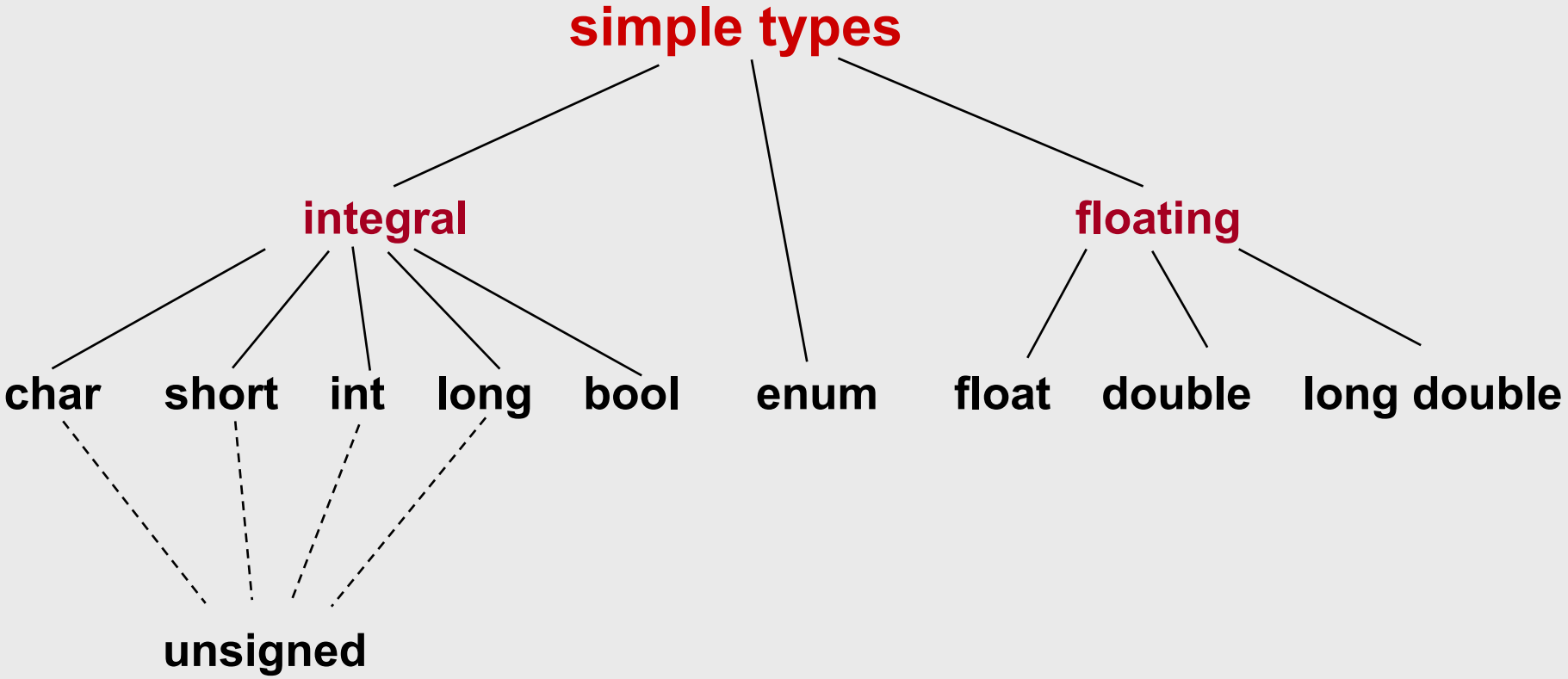
C++

- In C++, you can define local variables at any point in your program.
- In C++, you may define a global variable *only once*.
- *use of **static** is still supported, but it is not recommended for new code. Instead, you should use a namespace.*
- In C, you cannot find the address of a **register** variable using the **&** operator. But this restriction does not apply to C++.

C++ Data Types

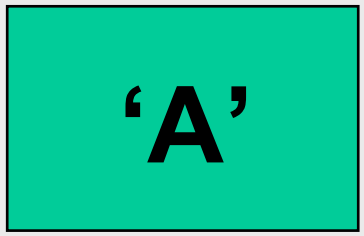

simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

C++ Simple Data Types



By definition,

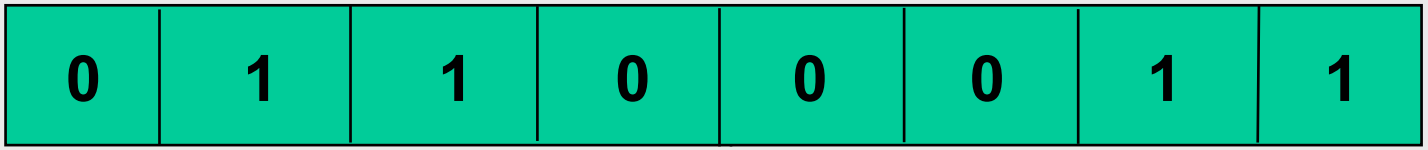
The size of a C++ char value is always 1 byte



exactly one byte of memory space

Sizes of other data type values in C++ are machine-dependent

Using one byte (= 8 bits)

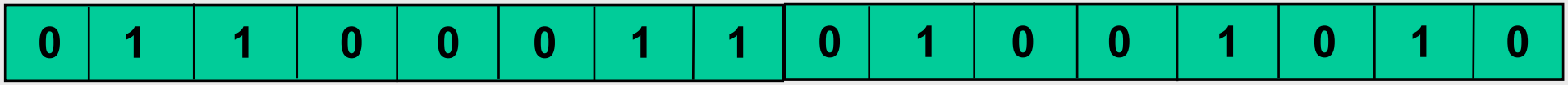


How many different numbers can be represented using 0's and 1's?

Each bit can hold either a 0 or a 1. So there are just two choices for each bit, and there are 8 bits.

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$$

Using two bytes (= 16 bits)



$$2^{16} = 65{,}536$$

So 65, 636 different numbers can be represented

If we wish to have only one number representing the integer zero, and half of the remaining numbers positive, and half negative, we can obtain the 65,536 numbers in the range -32,768 . . . . 0 32,767

Some Integral Types

Type	Size in Bytes	Minimum Value	Maximum Value
char	1	-128	127
short	2	-32,768	32,767
int	2	-32,768	32,767
long	4	-2,147,483,648	2,147,483,647

NOTE: Values given for one machine; actual sizes are machine-dependent

Data Type `bool`

- **Domain contains only 2 values, true and false**
- **Allowable operation are the logical (!, &&, ||) and relational operations**

Operator `sizeof`

sizeof A C++ unary operator that yields the size on your machine, in bytes, of its single operand. The operand can be a variable name, or it can be the name of a data type enclosed in parentheses.

```
int  age;
cout  <<  "Size in bytes of variable age is "
     <<  sizeof  age  <<  end;
cout  <<  "Size in bytes of type float is "
     <<  sizeof (float) <<  endl;
```

The only guarantees made by C++ are . . .

1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

1 <= sizeof (bool) <= sizeof (long)

sizeof (float) <= sizeof (double) <= sizeof (long double)

char is at least 8 bits

short is at least 16 bits

long is at least 32 bits

Floating Point Types

Type	Size in Bytes	Minimum Positive Value	Maximum Positive Value
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	1.1E+4932

NOTE: Values given for one machine; actual sizes are machine-dependent

More about Floating Point Types

- **Floating point constants in C++ like 94.6 without a suffix are of type double by default**
- **To obtain another floating point type constant a suffix must be used**
  - **The suffix F or f denotes float type, as in 94.6F**
  - **The suffix L or l denotes long double, as in 94.6L**

Header Files
climits and cfloat

- **Contain constants whose values are the maximum and minimum for your machine**
- **Such constants are FLT_MAX, FLT_MIN, LONG_MAX, LONG_MIN**

```
#include  <climits>
using  namespace  std;
            .
            .
            .
cout  << "Maximum long is "  << LONG_MAX
     << endl;
cout  << "Minimum long is "  << LONG_MIN
     << endl;
```

Combined Assignment Operators

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

A statement to add 3 to age

```
age  =   age + 3;
```

OR

```
age  +=  3;
```

A statement to subtract 10 from `weight`

```
int   weight;
cin >>  weight;
```

```
weight  =  weight - 10;
```

OR

```
weight  -=  10;
```

A statement to divide money by 5.0

```
float   money;
cin >>  money;
```

```
money  =  money / 5.0;
```

OR

```
money  /= 5.0;
```

A statement to double `profits`

```
float   profits;
cin >>  profits;
```

```
profits  = profits * 2.0;
```

OR

```
profits  *=  2.0;
```

A statement to raise `cost` 15%

```
float   cost;
cin >>  cost;
cost = cost + cost * 0.15;
```

OR

```
cost = 1.15 * cost;
```

OR

```
cost *= 1.15;
```

Which form to use?

- **When the increment (or decrement) operator is used in a "*stand alone*" statement to add one (or subtract one) from a variable's value, it can be used in either prefix or postfix form**


```
dogs--;
```

```
--dogs;
```

BUT...

- **when the increment (or decrement) operator is used in a statement with other operators, the prefix and postfix forms can yield *different* results**

Let's see how. . .

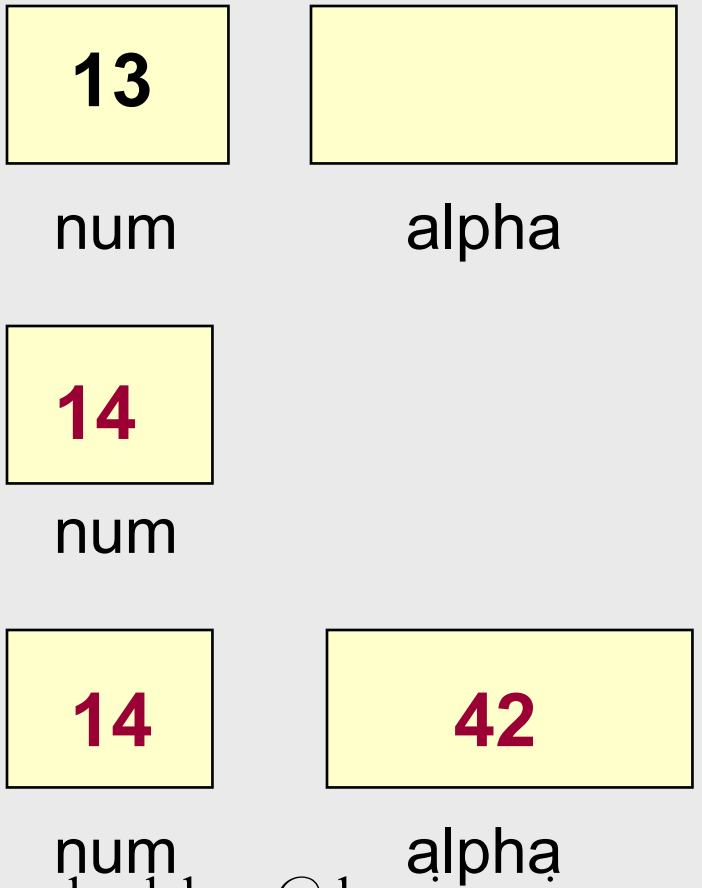
PREFIX FORM

"First increment, then use "

```
int  alpha;
int  num;

num = 13;


alpha = ++num * 3;
```

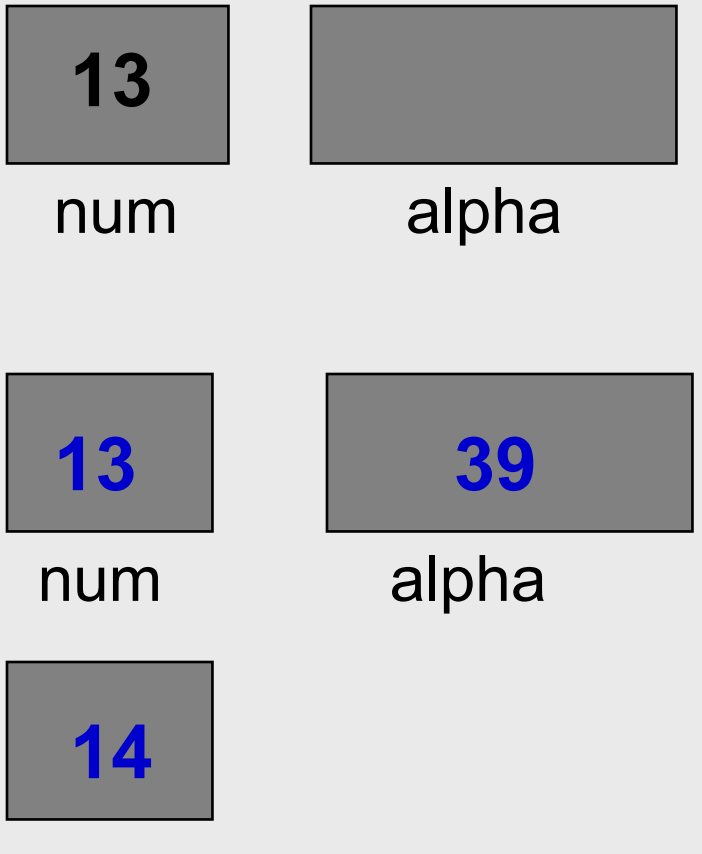


POSTFIX FORM
"Use, then increment"

```
int  alpha;
int  num;

num = 13;


alpha = num++   * 3;
```

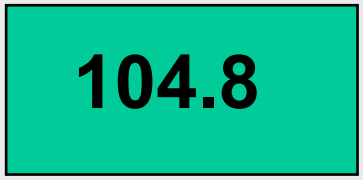


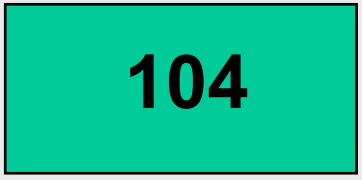
Type Cast Operator

The C++ cast operator, which comes in two forms, is used to explicitly request a type conversion

```
int    intVar;
float  floatVar  =  104.8;

intVar  =  int(floatVar);  // Functional notation, OR
intVar  =  (int)floatVar;  // Prefix notation uses ()
```





floatVar **intVar**

Ternary (three-operand) Operator
? :

SYNTAX

Expression1 ? Expression2 : Expression3

MEANING

If *Expression1* is true (nonzero), then the value of the entire expression is *Expression2*. Otherwise, the value of the entire expression is *Expression 3*.

For example . . .

Using Conditional Operator

```
float   Smaller (float  x,  float  y)
// Finds the smaller of two float values
// Precondition: x assigned  &&  y assigned
// Postcondition:Function value  ==   x, if x < y
//                                    ==  y, otherwise
{
   float   min;

   min  = (x < y)  ?   x :  y;
   return  min;
}
```

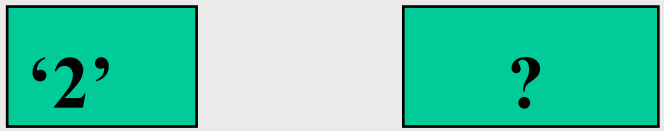
C++ Operator Precedence (highest to lowest)

Operator	*Associativity*
()	Left to right
unary: ++ -- ! + - (cast) sizeof	Right to left
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
||	Left to right
? :	Right to left
= += -= *= /=	Right to left

Converting `char` digit to `int`

- **The successive digit characters '0' through '9' are represented in ASCII by the successive integers 48 through 57 (the situation is similar in EBCDIC)**

- **As a result, the following expression converts a char digit value to its corresponding integer value**




```
char ch;
int  number;
. . .
number = int (ch - '0')
```

typedef statement

- **typedef creates an additional name for an already existing data type**
- **Before bool type became part of ISO-ANSI C++, a Boolean type was simulated this way**

```
typedef  int  Boolean;
const  Boolean  true = 1;
const  Boolean  false  = 0;
            .
            .
            .
Boolean dataOK;
            .
            .
            .
dataOK  =  true;
```

Enumeration Types

- **C++ allows creation of a new simple type by listing (enumerating) all the ordered values in the domain of the type**

EXAMPLE

```
enum MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                 JUL,  AUG,  SEP,  OCT,  NOV,  DEC };
```

name of new type

list of all possible values of this new type

enum Type Declaration

```
enum MonthType { JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC};
```

- **The enum declaration creates a new programmer-defined type and lists all the possible values of that type--any valid C++ identifiers can be used as values**
- **The listed values are ordered as listed; that is,**

  JAN < FEB < MAR < APR , and so on
- **You must still declare variables of this type**

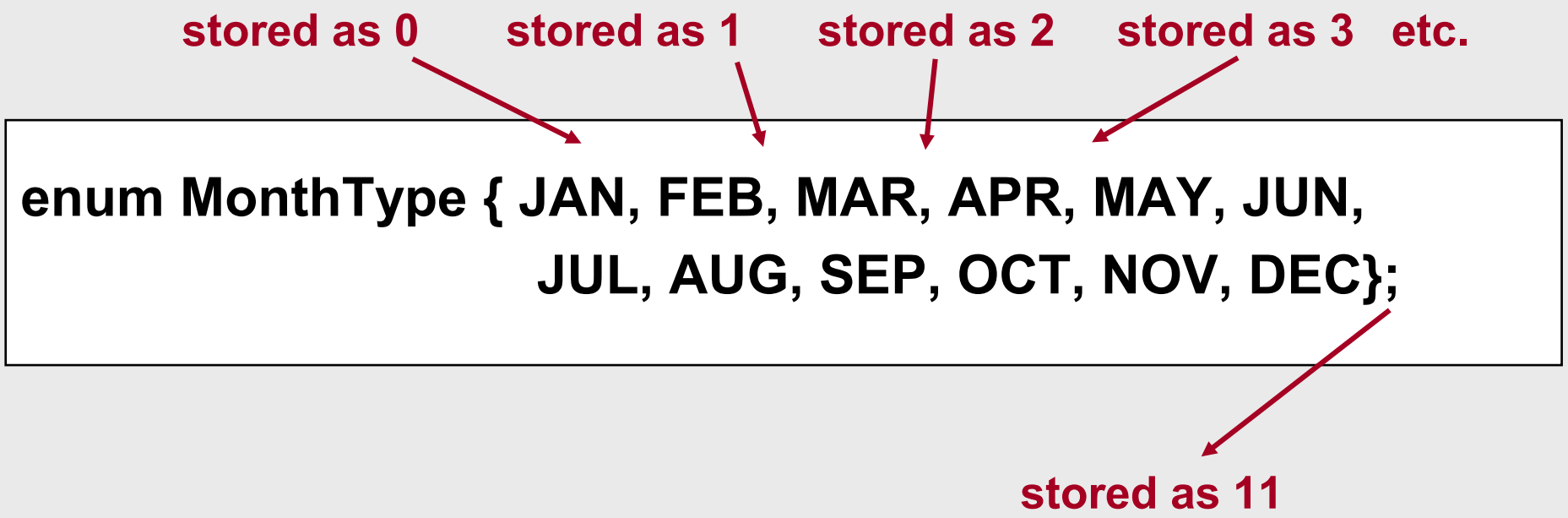
Declaring enum Type Variables

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                  JUL,  AUG,  SEP,  OCT,  NOV,  DEC };

MonthType  thisMonth; // Declares 2 variables
MonthType  lastMonth; // of type MonthType

lastMonth  =  OCT;    // Assigns values
thisMonth  =  NOV;    // to these variables
        .
        .
        .
lastMonth = thisMonth;
thisMonth = DEC;
```

Storage of enum Type Variables



Use Type Cast to Increment `enum` Type Variables

```
enum MonthType { JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC};
MonthType  thisMonth;
MonthType  lastMonth;

lastMonth  =  OCT;
thisMonth  =  NOV;
lastMonth = thisMonth;

thisMonth = thisMonth++; // COMPILE ERROR !

thisMonth = MonthType(thisMonth + 1);
// Uses type cast
```

More about enum Type

Enumeration type can be used in a Switch statement for the switch expression and the case labels

Stream I/O (using the insertion << and extraction >> operators) is not defined for enumeration types; functions can be written for this purpos

Comparison of enum type values is defined using the 6 relational operators (< , <= , > , >= , == , !=)

An enum type can be the return type of a value-returning function in C++

```
MonthType  thisMonth;

switch (thisMonth) // Using enum type switch expression
{
   case   JAN  :
   case   FEB  :
   case   MAR  :   cout << "Winter quarter";
                          break;
     case   APR  :
     case   MAY  :
     case   JUN  :   cout << "Spring quarter";
                          break;
     case   JUL  :
     case   AUG  :
     case   SEP  :   cout << "Summer quarter";
                          break;
     case   OCT  :
     case   NOV  :
     case   DEC  :   cout << "Fall quarter";
}
```

Using enum type Control Variable with for Loop

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                  JUL,  AUG,  SEP,  OCT,  NOV,  DEC };

void  WriteOutName (/* in */ MonthType); // Prototype
          .
          .
          .
MonthType  month;

for  (month = JAN; month <= DEC;
     month = MonthType (month + 1))
// Requires use of type cast to increment
{
    WriteOutName (month);
   // Function call to perform output
          .
          .
          .
}
```

```
void   WriteOutName ( /*  in  */  MonthType   month)
// Prints out month name
// Precondition:  month is assigned
// Postcondition: month name has been written out

{   switch (month)
    {
     case JAN : cout << " January ";   break;
          case FEB : cout << " February ";   break;
          case MAR : cout << " March ";      break;
          case APR : cout << " April ";      break;
          case MAY : cout << " May ";        break;
          case JUN : cout << " June ";       break;
          case JUL : cout << " July ";       break;
          case AUG : cout << " August ";     break;
          case SEP : cout << " September ";          break;
          case OCT : cout << " October ";    break;
          case NOV : cout << " November ";           break;
          case DEC : cout << " December ";           break;
    }
}
```

Function with enum Type Return Value

```
enum SchoolType {PRE_SCHOOL, ELEM_SCHOOL,
    MIDDLE_SCHOOL,  HIGH_SCHOOL,  COLLEGE };
    .
    .
    .
SchoolType   GetSchoolData (void)

// Obtains information from keyboard to determine level
// Postcondition: Return value == personal school level
{
   SchoolType  schoolLevel;
   int age;
   int lastGrade;
   cout  << "Enter age :  "; // Prompt for information
   cin  >> age;
```

```
   if  (age  <   6)
      schoolLevel = PRE_SCHOOL;

   else
   {
      cout
        <<  "Enter last grade completed in school: ";
      cin  >> lastGrade;
      if  (lastGrade  <  5)
         schoolLevel  =  ELEM_SCHOOL;
      else if  (lastGrade  <  8)
         schoolLevel  =  MIDDLE_SCHOOL;
           else if  (lastGrade  < 12)
              schoolLevel  =  HIGH_SCHOOL;
           else
              schoolLevel  =  COLLEGE;
   }
   return  schoolLevel;  // Return enum type value
}
```

Storage Class Specifiers

- There are four storage class specifiers supported by C:
  - extern
  - static
  - register
  - auto
- *C++ adds another storage-class specifier called* ***mutable.***

The Dot (.) and Arrow (−>) Operators

- In C, the **.** (dot) and the −>(arrow) operators access individual elements of structures and unions.
- In C++, the dot and arrow operators are also used to access the members of a class.

Nesting

- Standard C specifies that at least 15 levels of nesting must be supported. In practice, most compilers allow substantially more.
- More importantly, Standard C++ suggests that at least 256 levels of nested **if**s be allowed in a C++ program.
- Standard C specifies that a **switch** can have at least 257 **case** statements. Standard C++ recommends that *at least* 16,384 **case** statements be supported.

Declaring Variables within Selection and Iteration Statements

- In C++ (but not C), it is possible to declare a variable within the conditional expression of an **if** or **switch**, within the conditional expression of a **while** loop, or within the initialization portion of a **for** loop.
- A variable declared in one of these places has its scope limited to the block of code controlled by that statement.

A Sample C++ Program

```
#include <iostream>  //In older C++ #include<iostream.h>
using namespace std;
int main()
{
    int i;
    cout << "This is output.\n"; // this is a single line comment
    /* you can still use C style comments */
    // input a number using >>
    cout << "Enter a number: ";
    cin >> i;
    // now,  a number using <<
    cout << i << " squared is " << i*i << "\n";
    return 0;
}
```

Program Explained…

- (**<iostream>** is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that **<iostream>** is one of the new-style headers defined by Standard C++. New-style headers do not use the **.h** extension.

The next line in the program is

```
using namespace std;
```

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. A namespace creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library.

- Now examine the following line.

```
int main()
```

As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

- The next line contains two C++ features.

  cout << "This is output.\n"; // this is a single line comment

- In C++, the << has an expanded role. It is still the left shift operator, but when it is used as shown in this example, it is also an *output operator*. Assume that **cout** refers to the screen.) You can use **cout** and the **<<** to output any of the built-in data types, as well as strings
  of characters.

- Note that you can still use **printf()** or any other of C's I/O functions in a C++ program.

Comments

- C++ defines two types of comments.

  1. /*…..*/ First, you may use a C-like comment, which works the same in C++ as in C.

  2. //

     You can also define a single-line comment by using; whatever follows such a comment is ignored by the compiler until the end of the line is reached.

- Next, the program prompts the user for a number. The number is read from the keyboard with this statement:

```
cin >> i;
```

- In C++, the **>>** operator still retains its right shift meaning. However, when used as shown, it also is C++'s *input operator*. This statement causes **i** to be given a value read from the keyboard.
- The identifier **cin** refers to the standard input device, which is usually the keyboard.
- In general, you can use **cin >>** to input a variable of any of the basic data types plus strings.
- You are free to use any of the C-based input functions, such as **scanf(),**

- The program ends with this statement:

  return 0;

- This causes zero to be returned to the calling process (which is usually the operating system). This works the same in C++ as it does in C. Returning zero indicates that the program terminated normally.
- Abnormal program termination should be signaled by returning a nonzero value.

Closer Look at the I/O Operators

- When used for I/O, the << and >> operators are capable of handling any of C++'s built-in data types. For example, this program inputs a **float**, a **double**, and a string and then outputs them:

```
#include <iostream>
using namespace std;
int main()
{
    float f;
    char str[80];
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;
    cout << "Enter a string: ";
    cin >> str;
    cout << f << " " << d << " " << str;
    return 0;
}
```

- When you run this program, try entering **This is a test.** when prompted for the string.
- When the program redisplays the information you entered, only the word "This“ will be displayed. The rest of the string is not shown because the >> operator stops reading input when the first white-space character is encountered. Thus, "is a test" is

  never read by the program.

C++

- /* Incorrect in C. OK in C++. */

```
int f()
{
    int i;
    i = 10;
    int j; /* won't compile as a C program */
    j = i*2;
    return j;
}
```

Here is another example. This version of the program from the preceding section declares each variable just before it is needed.

```
#include <iostream>
using namespace std;
int main()
{
    float f;
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;
    cout << "Enter a string: ";
    char str[80]; // str declared here, just before 1st use
    cin >> str;
    cout << f << " " << d << " " << str;
    return 0;
}
```

Debate

- Which declarations are better?

- Declare all variables at the start of a block

or

- At the point of first use

Output Statements

SYNTAX

cout << *Expression* << *Expression* . . . ;

These examples yield the same output:

```
cout << "The answer is " ;
cout << 3 * 4 ;
```

```
cout << "The answer is " << 3 * 4 ;
```

No Default to int

- There has been a fairly recent change to C++ that may affect older C++ code as well as C code being ported to C++.
- The C language and the original specification for C++ state that when no explicit type is specified in a declaration, type **int** is assumed.
- However, the "default-to-int" rule was dropped from C++ a couple of years ago, during standardization.

- For example, in C and older C++ code the following function is valid.

```
func(int i)
{
return i*i;
}
```

- In Standard C++, this function must have the return type of **int** specified, as shown here.

```
int func(int i)
{
return i*i;
}
```

- As a practical matter, nearly all C++ compilers still support the "default-to-int" rule for compatibility with older code. However, you should not use this feature for new code because it is no longer allowed.

The bool Data Type

- C++ defines a built-in Boolean type called **bool**. At the time of this writing, Standard C does not. Objects of type **bool** can store only the values **true** or **false**, which are keywords defined by C++.

Old-Style vs. Modern C++

There are really two versions of C++.

- The first is the traditional version that is based upon Bjarne Stroustrup's original designs. This is the version of C++ that has been used by programmers for the past decade.
- The second is the new, Standard C++ that was created by Stroustrup and the ANSI/ISO standardization committee.

- The key differences between old-style and modern code involve two features:
  - new-style headers and
  - the **namespace** statement.

- The first version shown here reflects the way C++ programs were written using old-style coding.

```
/*
An old-style C++ program.
*/
#include <iostream.h>
int main()
{
return 0;
}
```

- Here is the second version of the skeleton, which uses the modern style.

```
/*
A modern-style C++ program that uses
the new-style headers and a namespace.
*/
#include <iostream>
using namespace std;
int main()
{
return 0;
}
```

- This version uses the new-style header and specifies a namespace.

- The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that may be mapped to files by the compiler, although they need not be.
- Since the new-style headers are not filenames, they do not have a **.h** extension. They consist solely of the header name contained between angle brackets.
- For example, here are some of the new-style headers supported by Standard C++.

  <iostream> <fstream> <vector> <string>
- The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.
- Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library.

Working with an Old Compiler

- Just replace

  #include <iostream>

  using namespace std;

  with

  #include <iostream.h>

- This change transforms a modern program into an old-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace** statement.

C++ Data Type String

- **a string is a sequence of characters enclosed in double quotes**
- **string sample values**

  "Hello" "Year 2000" "1234"
- **the empty string (null string) contains no characters and is written as ""**

Strings

• Besides using an array of characters, a special library called the Standard Template Library provides an alternative. You can declare a string using the string keyword.

•To use a string in your program, first include the string library using the using namespace keywords followed by std;.

•In your program, declare a variable starting with the word string followed by a valid name for the variable.

Here are examples:

string Continent;
string Company;

When requesting its value from the user, by default, the string identifier is used to get only a one-word variable.

Here is an example program that requests a first and last names:

```
#include <iostream>
#include <string>
 using namespace std;
int main()
{ string FirstName, LastName;
cout << "Enter first name: "; cin >> FirstName;
 cout << "Enter last name: ";
cin >> LastName; cout << "\n\nFull Name: " << FirstName << " " <<
LastName << "\n\n";
 return 0;
}
```

You can initialize a string variable of any length. One technique is to use the assignment operator and include the string in double-quotes. Here is an example:

```
string UN = "United Nations";

cout << "The " << UN << " is an organization headed by a Secretary General";
```

Here is an example program that requests strings of any length from the user:

```
#include <iostream>
 #include <string>
using namespace std;
 int main()
{
string MusicAlbum;
string TrackTitle;
cout << "Welcome to Radio Request where the listeners select their
songs:\n";
cout << "Type the album name: ";
getline(cin, MusicAlbum);
 cout << "Type the song title: ";
 getline(cin, TrackTitle);
cout << "\nNow for your pleasure, we will play: " << TrackTitle <<
"\nfrom the " << MusicAlbum << " wonderful album.\n\n";
 return 0;
}
```

Object Oriented Programming

- Object-oriented programming took the best ideas of structured programming and combined them with several new concepts.
- A program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected).
- For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program.

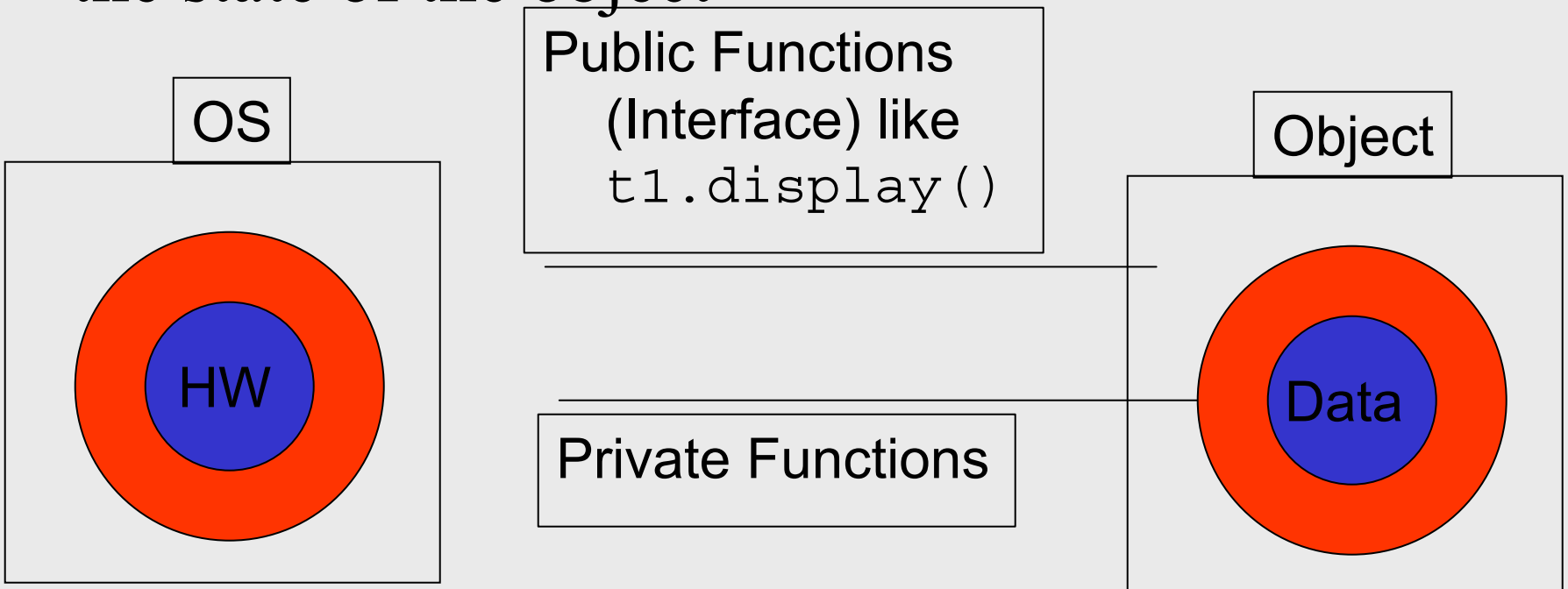
- Object-oriented programs are organized around data, with the key principle being "data controlling access to code."
- In an object-oriented language, you define the data and the routines that are permitted to act on that data.
- A data type defines precisely what sort of operations can be applied to that data.
- To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance.

Encapsulation

- *Encapsulation* is the mechanism that binds together code and the data it manipulates,and keeps both safe from outside interference and misuse.
- In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created.
- When code and data are linked together in this fashion, an *object* is created.
- In other words, an object is the device that supports encapsulation.

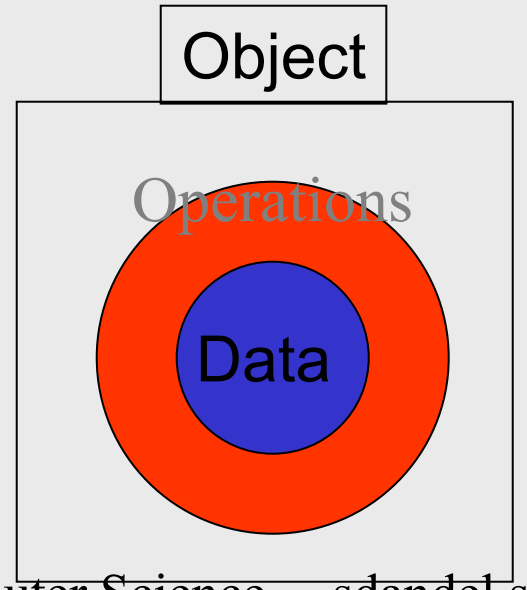
Information Hiding: a Design Principle

- HIDE the data from external access (and modification!) in order to protect the integrity of the state of the object



Encapsulation vs. Information Hiding

- Encapsulation is a language construct
- Information Hiding is a design principle
- Related but one is required the other is recommended



An object is like a black box.

The internal details are hidden.

- Identifying *objects* and assigning *responsibilities* to these objects.
- Objects communicate to other objects by sending *messages*.
- Messages are received by the *methods* of an object

- Within an object, code, data, or both may be *private* to that object or *public*.
- Private code or data may not be accessed by a piece of the program that exists outside the object.
- When code or data is public, other parts of your program may access it even though it is defined within an object.
- Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

Polymorphism

- Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods."
- A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees.
- It doesn't matter what type of furnace actually provides the heat.

Example…

- This same principle can also apply to programming. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push()** and **pop()** , that can be used for all three stacks.

- Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions.

- It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually.

Inheritance

- *Inheritance* is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications.

- For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*.

Inheritance Continued….

- Without the use of classifications, each object would have to define explicitly all of its characteristics.
- However, through the use of classifications, an object need only define those qualities that make it unique within its class.
- It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Introducing C++ Classes

- In C++, to create an object, you first must define its general form by using the keyword **class**.
- A **class** is similar syntactically to a structure.

Object Creation……

- Once you have defined a **class**, you can create an object of that type by using the class name. In essence, the class name becomes a new data type specifier.
- For example, this creates an object called **mystack** of type **stack**:

- stack mystack;

- The general form of a simple **class** declaration is:

class *class-name* {
private data and functions
public:
public data and functions
} *object name list*;

- When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member.

```
#define SIZE 100
// This creates the class stack.
class stack {
    private:
    int stck[SIZE];
    int tos;
    public:
    void init();
    void push(int i);
    int pop();
};
```

- For example, here is one way to code the **push()** function:

```
void stack::push(int i)
{
        if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
}
 stck[tos] = i;
 tos++;
}
```

Calling member functions and data…

- When you refer to a member of a class from a piece of code that is not part of the class, use the object's name, followed by the dot operator, followed by the name of the member.

- For example, this calls **init()** for object **stack1**.

  stack stack1, stack2;

  stack1.init();
- This fragment creates two objects, **stack1** and **stack2**, and initializes **stack1**.
- Understand that **stack1** and **stack2** are two separate objects.
- This means, for example, that initializing **stack1** does *not* cause **stack2** to be initialized as well.
- The only relationship **stack1** has with **stack2** is that they are objects of the same type.

- Within a class, one member function can call another member function or refer to a data member directly, without using the dot operator.
- It is only when a member is referred to by code that does not belong to the class that the object name and the dot operator must be used.
- Example: Stack

- The private members of an object are accessible only by functions that are members of that object. For example, a statement like

  stack1.tos = 0; // Error, tos is private.

  could not be in the **main()** function of the previous program because **tos** is private.

Classes

- Classes are created using the keyword **class**.
- A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class.
- A class is a logical abstraction, but an object has physical existence.
- An object is an *instance* of a class.
- A class declaration is similar syntactically to a structure.

Class Declaration

- class *class-name {*

  private data and functions

  access-specifier:

  data and functions

  access-specifier:

  data and functions

  // ...

  access-specifier:

  data and functions

  } object-list;

- The *object-list* is optional.
- *access-specifier* is one of these three C++ keywords:

  –public
  –private
  –Protected

  Variables that are elements of a class are called *member variables* or *data members*.

  No member can be declared as **auto**, **extern**, or **register**.

- By default, functions and data declared within a class are **private** to that class and may be accessed only by other members of the class.
- The **public** access specifier allows functions or data to be accessible to other parts of your program.
- The **protected** access specifier is needed only when inheritance is involved
- Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the

  class declaration is reached. Example….

- class employee {

```
char name[80]; // private by default
public:
void putname(char *n); // these are public
void getname(char *n);
private:
double wage; // now, private again
public:
void putwage(double w); // back to public
double getwage();
};
```

- Note: Actually, most programmers find it easier to have only one **private**, **protected**, and **public** section within each class.

- In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved.

- However, there may be situations in which you will need to make one or more variables public.

Differences

1. C does not provide classes;
   C++ provides both structs and classes

2. Members of a struct by default are **public** **(**can be accessed outside the struct by using the dot operator.)
   In C++ they can be declared to be **private** (cannot be accessed outside the struct.)

3. Members of a class by default are **private** (cannot be accessed outside the class) but can be explicitly declared to be **public**.

Structure and Classes

- The only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent.

Unions and Classes Are Related

- Like a structure, a **union** may also be used to define a class.
- In C++, **union**s may contain both member functions and variables.
- They may also include constructor and destructor functions.
- A **union** in C++ retains all of its C-like features, the most important being that all data elements share the same location in memory.
- Like the structure, **union** members are public by default and are fully compatible with C.

Anonymous Unions

- There is a special type of **union** in C++ called an *anonymous union*.
- An anonymous union does not include a type name, and no objects of the union can be declared.
- Instead, an anonymous union tells the compiler that its member variables are to share the same location.
- However, the variables themselves are referred to directly, without the normal dot operator syntax.

For example,

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
      {
      // define anonymous union
            union {
            long l;
            double d;
            char s[4];
            } ;
      // now, reference union elements directly
      l = 100000;
      cout << l << " ";
      d = 123.2342;
      cout << d << " ";
      strcpy(s, "hi");
      cout << s;
      return 0;
}
```

Function Overloading

- In C++, two or more functions can share the same name as long as their parameter declarations are different.
- In this situation, the functions that share the same name are said to be *overloaded*, and the process is referred to as *function overloading*

• To see why function overloading is important, first consider three functions defined by the C subset: **abs(), labs(),** and **fabs().** The **abs()** function returns the absolute value of an integer, **labs()** returns the absolute value of a **long**, and **fabs()** returns the absolute value of a **double**.

• Although these functions perform almost identical actions, in C three slightly different names must be used to represent these essentially similar

tasks.

•Example: Function Overloading

Constructors

- A *constructor function* is a special function that is a member of a class and has the same name as that of class.
- It is very common for some part of an object to require initialization before it can be used.
- This automatic initialization is performed through the use of a constructor function.
- An object's constructor is automatically called when the object is created.

- // This creates the class stack.
- class stack {

```
        int stck[SIZE];
        int tos;
        public:
        stack(); // constructor
        void push(int i);
        int pop();
        }; // stack's constructor function
stack::stack()
{
        tos = 0;
        cout << "Stack Initialized\n";
}
```

Destructor

- The complement of the constructor is the *destructor*.
- When an object is destroyed, its destructor (if it has one) is automatically called.
- There are many reasons why a destructor function may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor function that handles deactivation events.
- The destructor has the same name as the constructor, but it is preceded by a ~.

```
// This creates the class stack.
class stack {
   int stck[SIZE];
   int tos;
   public:
   stack(); // constructor
   ~stack(); // destructor
   void push(int i);
   int pop();
};
```

```
// stack's constructor function
stack::stack()
{
   tos = 0;
   cout << "Stack Initialized\n";
}
// stack's destructor function
stack::~stack()
{
   cout << "Stack Destroyed\n";
}
```

Full Example of constructor destructor:

Friend Functions

- It is possible to grant a nonmember function access to the private members of a class by using a **friend**.
- A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**.
- To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**.

```
#include <iostream>
using namespace std;
class myclass {
        int a, b;
        public:
        friend int sum(myclass x);
        void set_ab(int i, int j);
        };
void myclass::set_ab(int i, int j)
        {
        a = i; b = j;
        }
    // Note: sum() is not a member function of any class.
    int sum(myclass x)
    {
    /* Because sum() is a friend of myclass, it can
    irectly access a and b. */
    return x.a + x.b;
}
```

```
int main()
{
   myclass n;
   n.set_ab(3, 4);
   cout << sum(n);
   return 0;
}
```

- Full Example Friend Function

Friend Classes

- It is possible for one class to be a **friend** of another class.
- When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class.

```
// Using a friend class.
#include <iostream>
using namespace std;
class TwoValues {
        int a;
        int b;
        public:
        TwoValues(int i, int j) { a = i; b = j; }
        friend class Min;
        };
class Min {
        public:
        int min(TwoValues x);
        };
```

```
int Min::min(TwoValues x)
        {
        return x.a < x.b ? x.a : x.b;
        }

int main()
{
   TwoValues ob(10, 20);
   Min m;
   cout << m.min(ob);
   return 0;
}
```

Parameterized Constructors

- It is possible to pass arguments to constructor functions.
- Typically, these arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.

Parameterized Constructor Example

```
class myclass {
    int a, b;
    public:
    myclass(int i, int j) {a=i; b=j;}
    void show() {cout << a << " " << b;}
    };
int main()
{
    myclass ob(3, 5);
    ob.show();
    return 0;
}
```

- Specifically, this statement

  myclass ob(3, 4);

  causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass().**
- You may also pass arguments using this type of declaration statement:

  myclass ob = myclass(3, 4);

Constructors with One Parameter: A Special Case

```
#include <iostream>
using namespace std;
class X {
    int a;
    public:
    X(int j) { a = j; }
    int geta() { return a; }
};
int main()
{
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}
```

When Constructors and Destructors Are Executed

- An object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed.
- A local object's constructor function is executed when the object's declaration statement is encountered.
- The destructor functions for local objects are executed in the reverse order of the constructor functions.
- Global objects have their constructor functions execute *before* **main()** begins execution.
- Example:

Inline Functions

- In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation.
- This process is similar to using a function-like macro.
- To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword.

- //Inline Function

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
    {
    return a>b ? a : b;
    }
int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

Inline function Cont…….

- **Inline** functions may be class member functions.

Defining Inline Functions Within a Class

```
#include <iostream>
using namespace std;
class myclass {
        int a, b;
        public:
        // automatic inline
        void init(int i, int j) { a=i; b=j; }
        void show() { cout << a << " " << b << "\n"; }
        };
int main()
 {
        myclass x;
        x.init(10, 20);
        x.show();
        return 0;
}
```

The Scope Resolution Operator

```
int i; // global i
void f()
{
    int i; // local i
    i = 10; // uses local i
    .
    .
    .
}
```

- As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f()** needs to access the global version of **i**? It may do so by preceding the **I** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
    int i; // local i
    ::i = 10; // now refers to global i
    .
    .
    .
    }
```

Static Data Members

- When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable.
- No matter how many objects of a class are created, only one copy of a **static** data member exists.
- All **static** variables are initialized to zero before the first object is created.

- When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.)
- Instead, you must provide a global definition for it elsewhere, outside the class.
- This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs.
- Example:

Static Variable Cont……

- A **static** member variable exists *before* any object of its class is created. For example,

  in the following short program, **a** is both **public** and **static**. Thus it may

  be directly accessed in **main().** Further, since **a** exists before an object of **shared** is

  created, **a** can be given a value at any time.

```
#include <iostream>
using namespace std;
class shared {
    public:
    static int a;
    } ;
int shared::a; // define a
int main()
{
    // initialize a before creating any objects
    shared::a = 99;
    cout << "This is initial value of a: " << shared::a;
    cout << "\n";
    shared x;
    cout << "This is x.a: " << x.a;
    return 0;
}
```

Static Variable Uses:

- One use of a **static** member variable is to provide access control to some shared resource used by all objects of a class. Program:
- Another interesting use of a **static** member variable is to keep track of the number of objects of a particular class type that are in existence.

  Program:

Static Member Functions

- Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions.
- They may only directly refer to other **static** members of the class.
- A **static** member function does not have a **this** pointer.
- There cannot be a **static** and a non-**static** version of the same function.
- A **static** member function may not be virtual.
- Finally, they cannot be declared as **const** or **volatile**.

- Static function may be called either by itself, independent of any object, by using the class name and the scope resolution operator, or in connection with an object.

- Actually, **static** member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created.
- Example:

Nested Classes

- It is possible to define one class within another. Doing so creates a *nested* class.
- Since a **class** declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class.
- Nested classes are seldom used.
- Because of C++'s flexible and powerful inheritance mechanism, the need for nested classes is virtually nonexistent.

Local Classes(within a function)

- **When a class is declared within a function, it is known only to that function and unknown outside.**
- **All member functions must be defined within the class declaration.**
- **The local class may not use or access local variables of the function in which it is declared (except that a local class has access to static local variables declared within the function or those declared as extern).**
- **It may access type names and enumerators defined by the enclosing function.**
- **No static variables may be declared inside a local class.**

Passing Objects to Functions

- Objects may be passed to functions in just the same way that any other type of variable can.
- Objects are passed to functions through the use of the standard call-by-value mechanism.
- This means that a copy of an object is made when it is passed to a function. ie. another object is created.
- This raises the question of whether the object's constructor function is executed when the copy is made and whether the destructor function is executed when the copy is destroyed. Example:

Returning Objects

- A function may return an object to the caller.
- When an object is returned by a function, a temporary object is automatically created that holds the return value.
- It is this object that is actually returned by the function.
- After the value has been returned, this object is destroyed.
- The destruction of this temporary object may cause unexpected side effects in some situations.
- Example:

```
// Returning objects from a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
```

```
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

Object Assignment

- Assuming that both objects are of the same type, you can assign one object to another.
- This causes the data of the object on the right side to be copied into the data of the object on the left.

```
// Assigning objects.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
```

```
int main()
{
myclass ob1, ob2;
ob1.set_i(99);
ob2 = ob1; // assign data from ob1 to ob2
cout << "This is ob2's i: " << ob2.get_i();
return 0;
}
```

Arrays of Objects

- Object array is exactly the same as it is for any other type of array.

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
```

```
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

This program displays the numbers **1, 2,** and **3**

- If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays.

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};
```

```
int main()
{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen.

- If an object's constructor requires two or more arguments, you will have to use the longer initialization form.

```
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
cl(5, 6)
};
```

Pointers to Objects

- Just as you can have pointers to other types of variables, you can have pointers to objects.
- When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator.

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

Pointer Arithmetic

- All pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects.

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
```

```
int main()
{
cl ob[3] = {1, 2, 3};
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
}
```

Type Checking C++ Pointers

- There is one important thing to understand about pointers in C++: You may assign one pointer to another only if the two pointer types are compatible.
- For example, given:

```
int *pi;
float *pf;
```

in C++, the following assignment is illegal:

```
pi = pf; // error -- type mismatch
```

The this Pointer

- When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called).
- This pointer is called **this**.
- The **this** pointer is automatically passed to all member functions.

About this pointer

- **Friend** functions are not members of a class and, therefore, are not passed a **this** pointer.
- Second, **static** member functions do not have a **this** pointer.

Pointers to Derived Types

- Will be Covered Later after inheritance.

Pointers to Class Members

- C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object.
- This sort of pointer is called a *pointer to a class member* or a *pointer-to-member*, for short.
- A pointer to a member is not the same as a normal C++ pointer.

Pointers to Class Members Cont…..

- Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found.
- Since member pointers are not true pointers, the **.** and **->** cannot be applied to them.
- To access a member of a class given a pointer to it, you must use the special pointer-to-member operators **.*** and **–>*.**
- Example:

Contt….

- If you are using a pointer to the object, you need to use the –>* operator.
- Example:

References

- By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing.
- First, you can explicitly pass a pointer to the argument. Second, you can use a reference parameter.
- To create a reference parameter, precede the parameter's name with an **&**. Example:

Passing by value

```
void Math::square(int i)
{
    i = i*i;
}
int main()
{
    int i = 5;
    Math::square(i);
    cout << i << endl;
}
```

Passing by reference

```
void Math::square(int& i)
{
    i = i*i;
}
int main()
{
    int i = 5;
    Math::square(i);
    cout << i << endl;
}
```

What is a reference?

- An alias – another name for an object.

```
int x = 5;
int &y = x; // y is a
            // reference to x
y = 10;
```

- What happened to x?
- What happened to y?

What is a reference?

- An alias – another name for an object.

```
int x = 5;
int &y = x; // y is a
            // reference to x
y = 10;
```

- What happened to x?
- What happened to y? – y **is** x.

Why are they useful?

- Some people find it easier to deal with references rather then pointers, but in the end there is really only a syntactic difference (neither pass by value)
- Unless you know what you are doing do not pass objects by value, either use a pointer or a reference
- Can be used to return more than one value (pass multiple parameters by reference)

How are references different from Pointers?

Reference	Pointer
`int &a;`	`int *a;`

`int a = 10;`
`int b = 20;`
`int &c = a;`
`c = b;`
	`int a = 10;`
	`int b = 20;`
	`int *c = &a;`
	`c = &b;`

Returning References

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

- This program replaces the space between **Hello** and **There** with an **X**. That is, the program displays **HelloXthere**. Take a look at how this is accomplished. First, **replace()** is declared as returning a reference to a character. As **replace()** is coded, it returns a reference to the element of **s** that is specified by its argument **i.** The reference returned by **replace()** is then used in **main()** to assign to that element the character **X**.One thing to beware of when returning references is that the object being referred to does not go out of scope after the function terminates.

Independent References

- You candeclare a reference that is simply a variable. This type of reference is called an *independent reference*.
- All independent references must be initialized when they are created.

```
#include <iostream>
using namespace std;
int main()
{
int a;
int &ref = a; // independent reference
a = 10;
cout << a << " " << ref << "\n";
ref = 100;
cout << a << " " << ref << "\n";
int b = 19;
ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref--; // this decrements a
// it does not affect what ref refers to
cout << a << " " << ref << "\n";
return 0;
}
```

Output

- The program displays this output:

10 10

100 100

19 19

18 18

References to Derived Types

- Will be Covered Later after inheritance.

Restrictions to References

- You cannot reference another reference.
- Put differently, you cannot obtain the address of a reference.
- You cannot create arrays of references.
- You cannot create a pointer to a reference.
- You cannot reference a bit-field.
- A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value.
- Null references are prohibited.

Pointers

- In C, The code fragment shown here allocates 1,000 bytes of contiguous memory:

```
char *p;
p = malloc(1000); /* get 1000 bytes */
```

- After the assignment, **p** points to the start of 1,000 bytes of free memory.
- In C++, an explicit type cast is needed when a **void *** pointer is assigned to another type of pointer. Thus, in C++, the preceding assignment must be written like this:

```
p = (char *) malloc(1000);
```

- As a general rule, in C++ you must use a type cast when assigning (or otherwise converting) one type of pointer to another.

C++'s Dynamic Allocation Operators

- C++ provides two dynamic allocation operators: **new** and **delete**.
- These operators are used to allocate and free memory at run time.
- C++ also supports dynamic memory allocation functions, called **malloc()** and **free().** These are included for the sake of compatibility with C.
- However, for C++ code, you should use the **new** and **delete** operators because they have several advantages.

New and Delete

- The **new** operator allocates memory and returns a pointer to the start of it.
- The **delete** operator frees memory previously allocated using **new**.
- The general forms of **new** and **delete** are shown here:

  p_var = new *type*;

  delete *p_var*;
- Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Allocating memory using **new**

```
Point *p = new Point(5, 5);
```

- **`new`** can be thought of a function with slightly strange syntax
- **`new`** allocates space to hold the object.
- **`new`** calls the object's constructor.
- **`new`** returns a pointer to that object.

Deallocating memory using **delete**

```
// allocate memory
Point *p = new Point(5, 5);


...
// free the memory
delete p;
```

For every call to **`new`**, there must be exactly one call to **`delete.`**

Using **new** with arrays

```
int x = 10;
int* nums1 = new int[10];  // ok
int* nums2 = new int[x];   // ok
```

- Initializes an array of 10 integers on the heap.
- C++ equivalent of

```
int* nums = (int*)malloc(x *
sizeof(int));
```

Using **new** with multidimensional arrays

```
int x = 3, y = 4;
int* nums3 = new int[x][4][5];// ok
int* nums4 = new int[x][y][5];//
  BAD!
```

- Initializes a multidimensional array
- Only the first dimension can be a variable. The rest must be constants.
- Use single dimension arrays to fake multidimensional ones

Using **delete** on arrays

```
// allocate memory
int* nums1 = new int[10];
int* nums3 = new int[x][4][5];

...
// free the memory
delete[] nums1;
delete[] nums3;
```

- Have to use **`delete[]`**.

Limitations…

- Since the heap is finite, it can become exhausted.
- If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated.