Design and programming are human activities; forget that all is lost.
Bjarne Stroustrup, 1991

Language shapes the way we think, and determines what we can think about.
B.L. Whorf

2.1 Function Prototypes

Type and number of each argument must be specified in the declaration of the function, referred to as function prototype. In C++ empty argument list means that the function takes no arguments. Argument list with an explicit keyword void is recommended. In C++ failure to return a value in a function declared as returning a value is treated as an error. In C++ use of an undeclared function is an error. C++ casts to the appropriate type automatically. It permits better error detection at compile time.

2.1 Code listing

```
float x2 (float value)
{
   return (2 * value);
}

void main()
{
   int a = 10;
   float b = x2("1234");
}
```

Requires C header files to be modified. To C++ empty parentheses mean zero arguments.

```
extern float x2();        // wrong
extern float x2(float)    // right
```

Note

On **Unix** follow these steps for compilation

C program cc filename.c
C++ program CC filename.C (the command is upper case CC, and file extension is **.**C).

On **Linux** follow these steps for compilation

C program gcc filename.c
C++ program g++ filename.C (file extension is **.**C).

On **Turbo C++** follow these steps for compilation

C program tcc filename.c
C++ program tcc filename.cpp (file extension is **.**cpp).

On **Visual C++ (Microsoft C++)** follow these steps for compilation

C program cl filename.c
C++ program cl filename.cpp (file extension is **.**cpp).

2.2 Variable declarations

Declarations may appear anywhere within the body of a function. There is no extra runtime overhead as the compiler actually moves the declarations to the top of the block. It is more readable as the variable declarations are lexically closer to use.

```
int sumsq(int n)
{
   for(int j = 0, sum = 0; j < n; j++)
      sum += j*j;
   return sum;
}
```

2.3 Use Inline and const instead of #define

Prefer the compiler to the preprocessor, because #define is not part of the language. When we define something like

```
# define PI 3.1415
```

the symbolic name PI is never seen by the compiler; it is removed by the preprocessor before the source code ever gets to the compiler. As a result, the name PI doesn't get entered into the symbol table. This can be confusing if you get an error message during the compilation involving the use of the constant, because the error message will refer to 3.1415, not PI. If PI was defined in a header file you did not write, you would have no idea where that 3.1415 came from, and you would probably waste a lot of valuable time tracking it down.

2.3.1 Inline function declarations

Functions save memory space because all the calls to the function cause the same code to be executed. The functions need not be duplicated in memory. When the compiler sees a function call, it jumps to the function and at the end of the execution returns back to the calling program. This does save memory, but takes extra time. To save execution time in short functions (to make it run faster), the code is replaced in the function body directly inline with the code in the calling function (similar to 'C' macro). Functions, which generally contain 2-4 statements, are made inline. It replaces use of #define for macros. The **inline** specifier instructs the compiler to replace function calls with the code of the function body. This substitution is “inline expansion” (sometimes called “inlining”). Inline expansion alleviates the function-call overhead at the potential cost of larger code size. Any change in inline function would require all clients of the functions to be compiled. This can be significant in some program development and maintenance situations.

2.2 Code listing

```
inline int min(int x, int y)
{
   return(x < y ? x : y);
}
```

```
void main()
{
   int a = 10, b=25;
   int result = min(a, b);
}
```

2.3 Code listing

// Functions declared as inline can be passed around like regular functions.

```
#include <stdio.h>

inline int min(int x, int y)
{
   return(x < y ? x : y);
}

void main()
{
   int(*min_func)(int,int)=min;
   printf("%d\n", min_func(75,33));
}
```

The inline directive is a hint to the compiler, not a command (just like register). That means the compiler is free to ignore your inline directive whenever it wants to, and it is not that hard to make it want to. For example, most compilers refuse to inline functions that are recursive, and many of them won't inline complicated functions.

2.3.2 Constant type declarations

The const qualifier specifies that the value of a variable will not change throughout the program It replaces use of #define for constants. It allows you to communicate to both the programmer and the compiler that a particular value should remain invariant. The const keyword is very versatile. Outside of classes, you can use it for global constants and for static objects (local to either a file or a block). Inside classes you can use it for either static or non-static data members.

2.4 Code listing

```
void main()
{
   const x = 57;
   const float y = 12.5;
}
```

Can be used with data structures

```
   const complex c1(0,1);
```

Const structures can't be modified. Const structures can be passed to functions only if they declare the corresponding argument const

```
   void print (const complex *c);
```

2.5 Code listing

// A pointer to a constant can be used to view but not modify its target

```
char *buffer, *name;
void main()
{
   const  char *p = buffer;
   p = name;
   *p = 'x';            // illegal
}
```

// A const pointer can't be modified but its target can

```
char *buffer, *name;
void main()
{
   char* const q=buffer;
   q=name;            // illegal
   *q='x';
}
```

// We can also have constant pointers to constants

```
char *buffer, *name;
void main()
{
   const char* const r = buffer;
   r = name;          // illegal
   *r='x';            // illegal
}
```

2.4 Stream I/O Package

A stream is an abstraction that refers to the flow of data. The cout object, representing the standard output stream is a predefined object of the ostream_withassign class, which is derived from the ostream class. The standard output stream normally flows to the screen display, although it can be directed to other output devices. The operator << is called the 'insertion' or 'put to operator'. It directs the contents of the variable on its right to the object on its left. It is similar to the left-shift bit-wise operator of C language. This stream represents data coming from the keyboard. The operator >> is called the 'extraction' or 'get from operator'. It takes the value from the stream object on its left and places it in the variable on its right. The stream is an alternative to C's stdio package. The cerr object corresponds to the standard error stream which is used for displaying error messages. By default, this stream is associated with the standard output device, and the stream is unbuffered. Because the output is unbuffered, everything sent to cerr is written to the standard error device immediately. The clog object also corresponds to the standard error stream object except that the data is buffered. Stream I/O eliminates type mismatch between format strings and arguments which are there in printf() and scanf() calls. In scanf(), you have to pass an address if you don't have a pointer, and if you already have a pointer variable, you have to make sure you don't pass an address. Stream functions provide additional I/O operations that will be covered in detail later.

2.6 Code listing

```
# include <iostream.h>
# include <stdio.h>

void main()
{
   int i;
   char s[20];

   fprintf(stderr,"Enter an integer and string ");
   scanf("%d%s", &i, s);
   printf("\nOutput of printf and scanf Statement");
   fprintf(stderr, "\nInteger value is: %d ", i);
   printf("\nString value is: %s ", s);

   fflush(stdout);

   cerr << endl << "Enter an integer and string " << endl;
   cin >> i >> s;
   cout << " Output of cout and cin Statement" << endl;
   cout << endl << " Integer value is: " << i;
   cout << endl << " String  value is: " << s;
}
```

Note

We get different results on the screen with redirection and without redirection. Why? What is the effect of using cout instead of cerr or vice-versa?

2.5 Variable Reference

A reference holds the address of an object but behaves syntactically like an object. Used when it is necessary to change the actual arguments. When passing large class objects to a functions (under pass by value the entire object is copied with each call; as a reference, only the address of the argument is copied). Whenever a reference (or pointer) argument is not to be modified within the function, it is a good practice to declare the argument as const. This allows the compiler to prevent unintentional changes from occurring, especially in a chain of function calls to which the argument is passed.

A reference type is defined by following the type specifier with the address of operator. A reference object, like a constant, must be initialized.

```
   int val = 5;
   int &refval = val;  // ok
   int &refval2;       // error: uninitialized
```

A reference type, referred also as an alias, is an alternate name for the object with which it has been initialized. All operations applied to the reference act on the object to which it refers.

```
   refval += 10;       // adds 10 to val, setting it to 15
   int x = refval;     //   assigns   x   the   value   currently
associated with val
   int *pi = &refval;  // initializes pi with the address of val
```

The following initializations are an error

```
unsigned char ch;
double d1, d2;
int &ri = 100;          // error: not an lvalue
char &rc = ch;          // error: inexact type
double &d = d1 + d2;      // error: not an lvalue
```

Only a reference to a constant object or pointer can be initialized either with an rvalue, such as 100, or lvalue not of its exact type. The above declarations can be written as follows:

```
const int &ri = 100;
const char &rc = ch;
const double &d = d1 + d2;
```

A pointer to a constant can be used to view but not modify its target.

2.7 Code listing

// Code to illustrate passing parameters using address

```
#include <iostream.h>

void oldswap(int *x, int  *y)
{
   int temp = *x;
   *x =*y;
   *y=temp;
}

void main()
{
   int a = 5, b = 7;
   oldswap(&a, &b);
   cout << a << " " << b;
}
```

Permits function call by reference. The declaration of an argument as a reference overrides the default pass-by-value argument passing mechanism. The function receives the lvalue of the actual argument rather than a copy of the argument itself.

2.8 Code listing

```
#include <iostream.h>

void newswap(int &x, int &y)
{
   int temp = x;
   x = 7;
   y = temp;
}

void main()
```

```
{
   int a = 5, b = 7;
   newswap(a, b);
   cout << a << " " << b;
}
```

2.9 Code listing

// Also permits functions to return references (lvalue)

```
# include <stdio.h>

static int a[100] = {0};

int & subscript(int array[], int x, inty)
{
   return  array[x*10+y];
}

void main()
{
   subscript(a, 4, 3)=25;
   print("%d%d\n", a[43], subscript(a, 4, 3));
}
```

Restrictions that apply to reference variables are

2.10 Code listing

1) You cannot reference a reference variable (i.e. you cannot take its address)

```
# include <iostream.h>

void main()
{
   int a = 10;
   int &b = a;
   int &c = &b;   // illegal. If you comment, it will work.
   int &c = b;
   cout << " B is " << b << endl;
   cout << " C is " << c << endl;
}
```

2) You cannot create arrays of references.

```
void main()
{
   int a[5] = {1, 2, 3, 4, 5};
   int &b[5] = a     // illegal
   int &c = a;       // illegal
}
```

3) You cannot create a pointer to a reference.

```
# include <iostream.h>
```

```
void main()
{
   int a = 10,  *x = &a;
   int &b = a;
   // int *c = b;    // illegal
   int *c = &b;   // legal
   // int &d = x;    // illegal
   int &d = *x;   // legal

   cout<< "A is "<< a << endl;
   cout<< "B is "<< b << endl;
   cout<< "C is "<< *c << endl;
}
```

4) References are not allowed on bit fields.

```
# include <iostream.h>

struct demo
{
   int x:4;
};

void main()
{
   demo d;
   d.x = 10;
   int &b = d.x;
   cout << " b is " << b << endl;
}
```

2.5.1 Distinguish between pointers and references

Pointers and references look different enough (pointers use the "*" and "->" operators, references use "."), but they seem to do similar things. Both pointers and references let us refer to other objects indirectly. How do we decide when to use one and not other?

First recognize that there is no such thing as a null reference. A reference must always refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that variable is null, you should probably make the variable a reference.

Because a reference must refer to an object, C++ requires that references must be initialized.

```
   int & i;        // Error, reference must be initialized
   int j;
   int & i = j;    // OK, i refers to  j
```

Pointers are subject to no such restrictions

```
   int *k;
```

The fact that there is no such thing as a null reference implies that it can be more efficient to use references than to use pointers, because there's no need to test the validity of a reference before using it.

```
void printint (const int & j) {
   cout << j;// No need to test j, it must refer to a integer
}
```

Pointers, on the other hand should generally be tested against null

```
void printint(const int *j) {
   if(j)   cout << *j;
}
```

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. A reference, however, always refers to the object with which it is initialized.

2.11 Code listing

```
#include <iostream.h>

void main()
{
   int j = 20, k = 30;
   int & m = j;           // m refers to j
   int * n = &j;
   cout << j << " " << k << " " << m << endl;
   cout << &j << " " << &k << " " << &m << " " << n << endl;
   m = k;                 // m still points to j, but j's value is
now 30
   cout << j << " " << k << " " << m << endl;
   cout << &j << " " << &k << " " << &m << " " << n << endl;
   n = &k;                // n now points to k
   cout << j << " " << k << " " << m << " " << *n << endl;
   cout << &j << " " << &k << " " << &m << " " << n << endl;
}
```

In general, you should use a pointer whenever you need to take into account the possibility that there is nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points). You should use a reference whenever you know there will always be an object to refer to and you also know that once you are referring to that object, you will never want to refer to anything else.

2.6 Memory allocation operators

Memory allocation/deallocation in C++ is carried out using new and delete instead of malloc and free. New and delete know about the constructors and destructors. When new is called two things happen. First enough memory is allocated to hold an object of the type requested, and second, one or more constructors are called to initialize an object in the memory allocated.

New operator replaces most uses of 'malloc'.

- Accepts number of objects to be created
- Automatically calls initialization function
- Automatically returns proper pointer type

The new operator attempts to dynamically allocate (at run time) one or more objects of type-name. The new operator allocates from a program memory area called the “free store which is often referred to as the “heap.” When new is used to allocate a single object, it yields a pointer to that object. The new operator does not allocate reference types because they are not objects. If there is insufficient memory for the allocation request, by default operator new returns NULL.

Delete replaces most uses of 'free'

- Automatically calls cleanup function

Memory that is dynamically allocated using the new operator can be freed using the delete operator. The delete operator calls the operator delete function, which frees memory back to the available pool. Using the delete operator also causes the class destructor (if there is one) to be called.

2.12 Code listing

```
# include <stdio.h>

void main()
{
   const int numbercount = 5;
   int k, j, i;
   int *numbers = new int[numbercount];

   if(numbers != 0) {
      for(j = 0; j < numbercount;  j++)
         numbers[j] =  j  *  j;

      for(k = numbercount - 1; k >= 0; k--)
         printf("%3d%4d\n", k, numbers[k]);

      delete [] numbers;
   }
   else
      printf("Memory allocation failed\n");
}
```

When new can't allocate memory, it returns 0 or NULL, always check the return value of new, and take an action when memory allocation fails.

Try the following

Try the above program with the following statement. Why is the output different?

```
const int numbercount = -5;
```

2.7 Function Overloading

C++ allows specification of more than one function of the same name in the same scope. These are called “overloaded functions”. Overloaded functions enable programmers to supply different

semantics for a function, depending on the types and number of arguments. The compiler decides on the function to call based on the number of arguments and the argument types. C++ implements overloading by mangling functions names and argument lists into unique names. Overloading permits uniform naming and prevents programmers from having to invent names.

The overloading considerations are number of arguments, type of arguments, presence or absence of ellipsis and const or volatile. What is not considered in overloading are function return type, use of typedef names and unspecified array bounds.

Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call. If a suitable function is found, that function is called.

“Suitable” in this context means one of the following:

- An exact match was found.
- A trivial conversion was performed.
- An integral promotion was performed.
- A standard conversion to the desired argument type exists.
- A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.
- Arguments represented by an ellipsis were found.

The compiler creates a set of candidate functions for each argument. Candidate functions are functions in which the actual argument in that position can be converted to the type of the formal argument. A set of “best matching functions” is built for each argument, and the selected function is the intersection of all the sets. If the intersection contains more than one function, the overloading is ambiguous and generates an error. The function that is eventually selected is always a better match than every other function in the group for at least one argument. If this is not the case (if there is no clear winner), the function call generates an error.

Ambiguity of overloaded functions cannot be determined until a function call is encountered. At that point, the sets are built for each argument in the function call, and you can determine whether an unambiguous overload exists. This means that ambiguities can remain in your code until they are evoked by a particular function call.

2.13 Code listing

// Find the minimum of n numbers

```
# include<iostream.h>
int min(int x, int y)
{
   return(x < y ? x : y);
}

double min(double x, double y)
{
   return(x < y) ? x : y;
}

int min(int x, int y, int z)
```

```
{
   return min(x, min(y, z));
}

void main()
{
   int a = min(10, 3);
   double b = min(7.7, 13.3);
   int c = min(13, 10, 15);

   cout << " A " << a << " B " << b << " C " << c << endl;
}
```

2.14 Code listing

// Example- Displays specified number of characters

```
# include <iostream.h>

void repchar()
{
   for (int j = 0; j < 5; j++)
      cout << '*' << endl;
}

void repchar (char ch)
{
   for (int j = 0; j < 5; j++)
      cout << ch << endl;
}

void repchar (char ch, int  n)
{
   for(int j = 0; j < n; j++)
      cout << ch << endl;
}

void main()
{
   repchar();
   repchar('=');
   repchar('+',10);
}
```

2.7.1 Default argument functions

In the section on function overloading we had different versions of functions to do a set of similar operations. Default arguments are useful when the program is written and then the programmer decides to increase the capability of a function by adding another argument (i.e. the existing function calls can continue to use the old number of arguments while new function calls can use additional arguments).
In many cases, functions have arguments that are used so infrequently that a default value would suffice. To address this, the default-argument facility allows for specifying only those arguments to a function that are meaningful in a given call.

Now we are going to write a single function with default arguments for printing characters (the previous repchar function). In this example, the function is called 3 times from the main program. The first time it is called with no arguments, the second time with one argument & the third time with 2 arguments. The first two calls work because the called function provides default arguments.

The following points have to be considered when using default arguments:

- Default arguments are used only in function calls where trailing arguments are omitted - they must be the last argument(s). Therefore, the following code is illegal:
  ```
  void repchar (char = '*', int);
  ```
- A default argument cannot be redefined in later declarations even if the redefinition is identical to the original. Therefore, the following code produces an error:
  ```
  // Prototype for repchar function.
  void repchar(char = '*', int = 5)

  // Definition for repchar function.
  void repchar(char ch, int n = 2)
  {
  ...
  }
  ```
  The error generated is as follows: Redefinition of default parameter 2
- Additional default arguments can be added by later declarations.
  Default arguments can be provided for pointers to functions. For example:

2.15 Code listing

```
# include <iostream.h>

void repchar(char = '*', int = 5);

void main()
{
   repchar();
   repchar('=');
   repchar('+', 8);
}

void repchar(char ch, int n)
{
   for(int j = 0; j < n; j++)
      cout << ch << endl;
}
```

2.8 extern “C” keyword

C++ uses the same calling convention and parameter-passing techniques as C, but naming conventions are different because of C++ decoration of external symbols. By causing C++ to drop name decoration, the extern "C" syntax makes it possible for a C++ module to share data and routines with other languages.

Use of extern "C" has some restrictions:

- You cannot declare a member function with extern "C".
- You can specify extern "C" for only one instance of an overloaded function; all other instances of an overloaded function have C++ linkage.

```
// single statement linkage directive
extern "C" void exit(int);

// compound statement linkage directive
extern "C" {
   int printf(const char *,  …);
   int scanf(const char *,  …);
}

// compound statement linkage directive
extern "C" {
   #include <cmath.h>
}
```

2.16 Code listing

```
// add.c

int add(int x, int y)
{
   return x+y;
}

// test.cpp

# include <stdio.h>

extern "C" int add(int, int);

int add(int a, int b, int c)
{
   return a+b+c;
}

int main(int argc, char *argv[])
{
   printf("Add of 2 numbers is %d \n", add(5, 10));
   printf("Add of 3 numbers is %d \n", add(5, 10, 15));
   return 0;
}
```

Compile the above two files (add.c and test.cpp) using any C++ compiler. Link the files to create an exe (test.exe). When we execute the program we get the output as follows:

```
Add of 2 numbers is 15
Add of 3 numbers is 30
```

Try the following

1. Test the above program with the extern statement removed, i.e.
   modify the statement
   extern "C" int add(int, int);
   to
   int add(int, int);
   Files to be compiled are add.c and test.cpp.

   What is the cause of error during the linking process?

2. Rename file add.c to add.cpp. Compile the file test.cpp with the statement
   extern "C" int add(int, int);

   Files to be compiled are add.cpp and test.cpp.

   Why is there an error during linking?

3. Rename file add.c to add.cpp. Compile the file test.cpp without the extern keyword
   int add(int, int);

   Files to be compiled are add.cpp and test.cpp.

   Why does it work now?

2.9 Operator overloading

Permits extension of standard 'C" operators to support user defined data types. Using the concept of operator overloading we would make the user defined data types behave exactly like the pre-defined types in C++. We will see examples of this in the next chapter.

2.9.1 Rules for Overloaded operators

- Only operators that already exist can be overloaded i.e. you can't create new operators like $ or @. Most operators can be overloaded

+	-	*	/	%	^	&	|	~	!	=	<	>
<=	>=	++	_	<<	>>	==	!=	&&	||	+=	.=	*=
/=	%=	^=	|=	<<=	>>=	[]	()	->	new	delete		

- You cannot change an operator's template. Every C++ operator comes with its own template that defines certain aspects of its use such as whether it's binary or a unary operator, its place in order of precedence. Thus ++ and -- can never be used other than unary operator.

- Operators can only be overloaded when used with abstract data class. It isn't possible to redefine an operator in a situation where no user-defined classes are involved. i.e. you can't change the way in which ++ works with integers.

- Unary operators overloaded by means of a member functions takes no explicit arguments and return no explicit values, unary operators overloaded by means of a friend function take one reference argument, namely the name of the relevant class.

- Binary operators overloaded by means of a member function take an explicit argument; binary operators overloaded by means of a friend function take two explicit reference arguments. Binary operators such as +, -, * and / most explicitly return a value; they must not change their own arguments. Num3 = num1 + num2

Note

Never give an overloaded operator a meaning that is radically different from its natural meaning. Although it might be possible to overload the +operator to perform multiplication it is not advisable, as it would render the resulting code unreadable.

2.10 Variable arguments function

The compiler supports stdarg.h library that contains several functions & macros, which are used in conjunction with functions taking unspecified number of arguments. In the example below we are declaring a function print of variable number of arguments (specified by ellipsis operator in the function declaration). The ellipses syntax (**…**) turns off compiler type checking. The type va_list is used to declare a variable that will refer to each argument in turn.

2.17 Code listing

```
# include <iostream.h>
# include <stdarg.h>

void print(char *fmt, ...)
{
   va_list ptr;
   char *p, *sval;
   int ival;
   double dval;

   va_start(ptr, fmt);

   for(p = fmt; *p; p++)
   {
      if(*p != '%')
      {
         cout << (*p);
         continue;
      }

      switch(*++p)
      {
         case 'd':
            ival = va_arg(ptr, int);
            cout << ival;
            break;

         case 'f':
            dval = va_arg(ptr, double);
            cout << dval;
            break;
```

```
         case 's':
            for(sval = va_arg(ptr, char*); *sval; sval++)
            cout << *sval;
            break;

         default :
            cout << *p;
            break;
      }
   }
   va_end(ptr);
}

void main()
{
   int i = 10;
   char s[20] = "genesis";
   double f=12.12;
   print("%d\n", i);
   print("%s\n", s);
   print("%f\n", f);
}
```

In the variable argument function print() the first argument is obligatory; this is a character pointer (in the above program), and it's used to inform print() when to stop processing (it will stop when the end of string is encountered). The variable ptr is declared to be of special type va_list. Variable ptr is then passed to the library functions va_start(), va_arg() & va_end(). The macro va_start() is called with arguments ptr & fmt; this initializes ptr to point to the beginning of the list (first unnamed argument). It must be called before ptr is used. There must be atleast one named argument; the final named argument is used by va_start to get started. The for loop then uses the function va_arg() to extract one argument from the list at a time & prints the same. It steps ptr to point to the next argument; it uses a type name to determine what type to return and how big a step to take. The function va_end() then closes the operation and does all the cleanup necessary, and it must be called before the function returns.

2.12 Exercise

Theory Questions

1. Explain the concept of overloading in C++?
2. Write notes on Streams.
3. #define has become obsolete in C++. Justify?
4. What is the use of extern 'C' declaration?
5. Function prototyping helps in compile time error detection. How?
6. How inline functions are different from #define macros?
7. What is the principle reason for passing arguments by reference?
8. What is the default argument function? What are its limitations?
9. What is scope of a variable defined within a block?
10. What is the most important role of a function?

11. What is a function argument?
12. What are overloaded functions?
13. Why do we need the preprocessor directive #include <iostream.h>
14. Describe major parts of a C++ program?
15. In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
16. How does a constant defined by const differ from the constant defined by the preprocessor statement #define?
17. Why does C++ have type modifiers?
18. What do you mean by dynamic initialization of a variable? Give an example?
19. What is a reference variable? What is the use of it?
20. List at least four new operators added by C++ that aids OOP?
21. What is the advantage of new operator compared to function malloc()?
22. What is the significance of empty parenthesis in a function declaration?
23. How are variable arguments function implemented in C++?
24. What is a self-referential structure? Give some examples of self-referential structures?
25. What is the difference between a structure and union in 'C'?

Problems

1. Write a macro that prints the largest of three numbers?

2. Write a program that removes C++ comments from a given file and outputs the file data to console. The comment style should include both // comments and /* */.

3. Write a function cat() that takes two strings as arguments and returns a string that is concatenation of the arguments. Use new to allocate memory for the result.

4. Write a function rev() that takes a string argument and reverses the characters in it. Use this to check if a given string is a palindrome.

5. Write a function power() to raise a number M to a power N. The function takes a double value for M and int value for N and returns the result correctly. Use a default value of 2 for N to make the function calculate the square when the argument is omitted. Write a main that reads the values of M and N from the user to test the function.

6. Write a program "greetings" that takes a name as a command line argument and writes "Hello name". Modify this program to take any number of names as arguments and say hello to each.

7. Write a C++ program to convert temperature from Fahrenheit to Celsius (0 - 400 degrees). Use the following formula Celsius = (5.0 / 9.0) * (Fahrenheit - 32.0).

8. Write a C++ program to find the largest and smallest number from an array of 10 numbers?

9. Write a variable argument function to find the sum of all numbers (int) passed as argument. The first argument is named argument which refers to the number of variable arguments which follow (Example: sum(5, 1, 4, 7, 8, 23), should return the sum of 1, 4, 7, 8 and 23).

10. Write a recursive function to find the product of all numbers upto N. Accept the input data using command line arguments.