8.8 Structures and Functions

Structure variables may be passed to functions just like any other variables. It is also possible for functions to return structure variables through the use of the `return` statement. Note that any number of structure variables can be passed to the function as arguments in the function call, but only one structure variable can be returned from the function by the return statement. The program `student5.cpp` illustrates the passing of structure parameters and returning of a structure value.

```
// student5.cpp: structure data type parameter passing
#include <iostream.h>
struct Student
{
   int roll_no;
   char name[25];
   char branch[15];
   int marks;
};
// reads data of type Student and returns
Student read()
{
   Student dull;
   cout << "Roll Number ? ";
   cin >> dull.roll_no;
   cout << "Name ? ";
   cin >> dull.name;
   cout << "Branch ? ";
   cin >> dull.branch;
   cout << "Total Marks <max-325> ? ";
   cin >> dull.marks;
   return dull;   // returning structure variables
}
// displays contents of the structure Student
void show( Student genius )   // takes structure type parameter
{
   cout << "Roll Number: " << genius.roll_no << endl;
   cout << "Name: " << genius.name << endl;
   cout << "Branch: " << genius.branch << endl;
   cout << "Percentage: " << genius.marks*(100.0/325) << endl;
}
void main()
{
   // data definitions of 10 students
   Student s[10];
   int n;
   cout << "How many students to be processed <max-10>: ";
   cin >> n;
   // read student data
   for( int i = 0; i < n; i++ )
   {
      cout << "Enter data for student " << i+1 << "..." << endl;
```

```
      s[i] = read();
   }
   cout << "Students Report" << endl;
   cout << "---------------" << endl;
   // process student data
   for( i = 0; i < n; i++ )
      show( s[i] );
}
```

Run

```
How many students to be processed <max-10>: 2
Enter data for student 1...
Roll Number ? 3
Name ? Smrithi
Branch ? Genetics
Total Marks <max-325> ? 295
Enter data for student 2...
Roll Number ? 10
Name ? Bindhu
Branch ? MCA
Total Marks <max-325> ? 300
Students Report
---------------
Roll Number: 3
Name: Smrithi
Branch: Genetics
Percentage: 90.7692
Roll Number: 10
Name: Bindhu
Branch: MCA
Percentage: 92.3077
```

Passing Structure to a Function

In `main()`, the statement

```
show( s[i] );
```

passes a parameter of type structure `Student` to `show()` using the *pass-by-value* mechanism. All the members of the structure `s[i]` are assigned to respective members of the formal structure-parameter `genius` in the function `show()`. Any modification to the members of the structure variable `genius` in `show()` will not be reflected in the actual parameter `s[i]`.

Returning Structure from Function

Similar to variables of the standard data types, a variable of a structure type can be assigned to another variable of the same type. It is performed by using the assignment operator, which copies all the members by a one-to-one correspondence.

In `main()`, the statement

```
s[i] = read();
```

invokes `read()` and assigns all the members of a structure returned by `read()` to the structure

variable s[i]. Here all the members are copied to the destination variable on a member-by-member basis as shown in Figure 8.10.

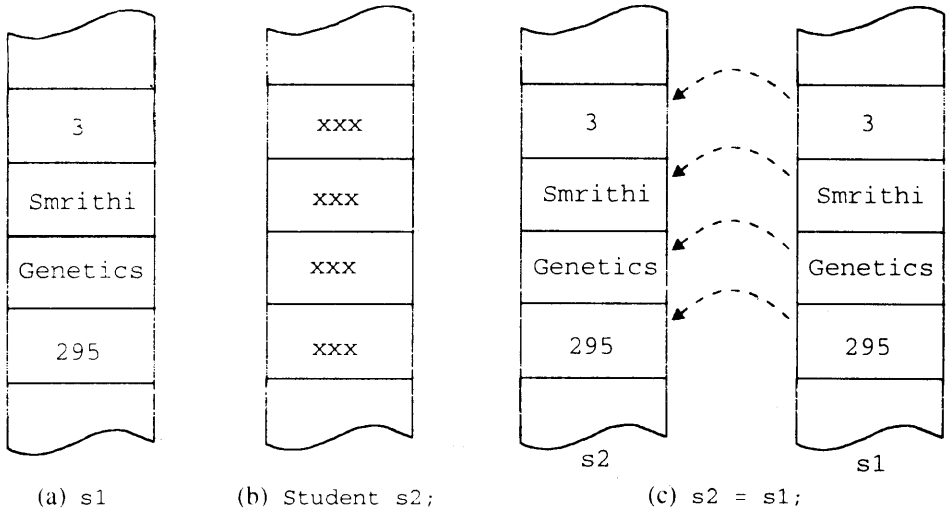


Figure 8.10: Structure assignment -- copied on a member-by-member basis

Passing an Array of Structures to Functions

Passing an array of structures to functions involves the same syntax and properties like passing any array to a function. Pass by reference method is employed and consequently, any changes made to the structures by the function are visible throughout the program. The program `student6.cpp` illustrates the passing an array of structures to a function.

```
// student6.cpp: passing array of structures
#include <iostream.h>
struct Student
{
   int roll_no;
   char name[25];
   char branch[15];
   int marks;
};
// return index to a structures which holds student details who
// scores highest marks in the university examination
int HighestMarks( Student s[], int count )
{
   int index, big;
   big = s[0].marks;
   index = 0;
   for( int i = 1; i < count; i++ )
   {
      if( s[i].marks > big )
      {
         big = s[i].marks;
```

```
            index = i;
         }
      }
      return index;
}
// reads data of type Student and returns
Student read()
{
   Student dull;
   cout << "Roll Number ? ";
   cin >> dull.roll_no;
   cout << "Name ? ";
   cin >> dull.name;
   cout << "Branch ? ";
   cin >> dull.branch;
   cout << "Total Marks <max-325> ? ";
   cin >> dull.marks;
   return dull;   // returning structure variables
}
// displays contents of the structure Student
void show( Student genius )   // takes structure type parameter
{
   cout << "Roll Number: " << genius.roll_no << endl;
   cout << "Name: " << genius.name << endl;
   cout << "Branch: " << genius.branch << endl;
   cout << "Percentage: " << genius.marks*(100.0/325) << endl;
}
void main()
{
   // data definitions of 10 students
   Student s[10];
   int n, id;
   cout << "How many students to be processed <max-10>: ";
   cin >> n;
   // read student data
   for( int i = 0; i < n; i++ )
   {
      cout << "Enter data for student " << i+1 << "..." << endl;
      s[i] = read();
   }
   id = HighestMarks( s, n );
   cout << "Details of student scoring highest marks..." << endl;
   show( s[id] );
}
```

Run

```
How many students to be processed <max-10>: 3
Enter data for student 1...
Roll Number ? 3
Name ? Smrithi
```

```
Branch ? Genetics
Total Marks <max-325> ? 295
Enter data for student 2...
Roll Number ? 15
Name ? Rajkumar
Branch ? Computer
Total Marks <max-325> ? 315
Enter data for student 3...
Roll Number ? 7
Name ? Laxmi
Branch ? Electronics
Total Marks <max-325> ? 255
Details of student scoring highest marks...
Roll Number: 15
Name: Rajkumar
Branch: Computer
Percentage: 96.9231
```

In `main()`, the statement

```
id = HighestMarks( s, n );
```

invokes the function `HighestMarks()` and finds the student with the highest marks. It accepts two arguments, the first is an array of structures and the second argument is an integer which denotes the number of students. The index of the student record with the highest marks is found by this function and returned to its caller (in this case, `main()` is the caller).

8.9 Data Type Enhancement Using `typedef`

C++ provides a facility called type definition by which new type names can be created. This is accomplished by using the `typedef` keyword as shown in Figure 8.11.

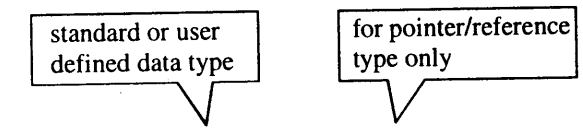


Figure 8.11: Enhancing existing data types

`ExistingTypeName` is the name of an existing data type, and `NewTypeName` is the new user defined data type. Notice that a new user defined data type is created only from the existing data types such as `int`, `float`, `struct`, etc. The following examples illustrate the concepts introduced.

```
typedef int Length;
```

Length now becomes a synonym for `int` and variables can be defined using the new type name. `Length` denotes a type name like `int` and is not a variable. Consider the following statement:

```
Length len1, len2;
```

The above statement defines two variables of type integer and is equivalent to

```
int len1, len2;
```

Note that the operations possible on the variables `len1` and `len2` are precisely the same as the operations permitted on integer variables defined using the keyword `int`. Consider the following set of statements.

```
typedef int emprec[10];
emprec person1, person2;
```

The type `emprec` is now a new data type which is a 10 element array of integer quantities. `person1` and `person2` are two variables of this new type and each variable is a 10 element array of integer quantities. The following are valid expressions:

`person1[3]`	access the 4th element of person1
`person1`	access the starting address of person1
`&person1[0]`	access the starting address of person1

The `typedef` statement for defining string data type is

```
typedef char * String;
```

It can be used as follows:

```
String name;
```

It is equivalent to

```
char * name;
```

The `typedef` can be used to create reference type (alias) integer data type as follows:

```
typedef int & INTREF;
```

Aliases for variables can be created using `INTREF` as follows:

```
INTREF b = c;
```

It is effectively equivalent to

```
int &b = c;
```

Benefits of the typedef statement

There are several important uses of the `typedef` statement:

- It helps in effective documentation of a program, thus increasing its clarity. This in turn enhances the ease of maintenance of the program, which is an important part of software management.
- The `typedef` statement is often used for declaring new data types involving structures. A new data type representing the structure is declared using the `typedef` keyword. Since all structure declarations in C++ are `typedef` by default, explicit use of the `struct` keyword during structure variable definition is optional. It is used explicitly when the structure's pointer or alias type is to be created. The usage of the `typedef` statement is illustrated below:

```
typedef struct tag
{
   type member1;
   type member2;
   ...
   type membern;
} [*/&] NewDataType;
```

Consider the following declarations:

```
struct date
{
```

```
    int day;
    int month;
    int year;
};
typedef date * DATEPTR;
```

The type name DATEPTR can be used to define a pointer to the structure date as follows:

```
DATEPTR dp;
```

It is equivalent to

```
date * dp;
```

- The third important use of the typedef statement is its usage in writing portable programs. The sizes of different data types are dependent on the compiler. For instance, the size of an integer is two bytes on a 16-bit compiler and four bytes on a 32-bit compiler. Portability is achieved by type-declaring an integer as follows:

```
typedef long int INT;
```

In the program, use definitions such as

```
INT a, b;
```

instead of the statement

```
int a, b;
```

to increase the portability of a program.

8.10 Structures and Encapsulation

Structures in C++ have undergone a major revision. Like C structures, C++ structures also provide a mechanism to group together data of different types into a single unit. In addition to this, C++ allows to associate functions as part of a structure. Thus, C++ structures provide a true mechanism to handle data abstraction. Such structures have two types of members: data members and member functions. (See Figure 8.12) Functions defined within a structure can operate on any member of the structure.

The program complex.cpp illustrates the concept of associating functions operating on the structure members. The functions enclosed within a structure can access data or other member functions directly. Similar to the data members, member functions can be accessed using the dot operator.

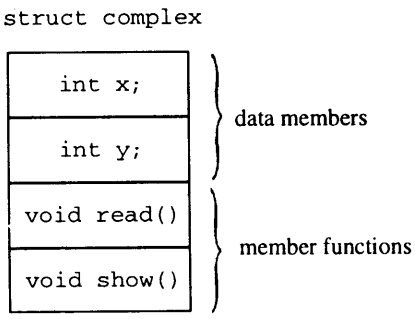


Figure 8.12: Functions as a part of C++ structures

```
// complex.cpp: functions as a part of C++ structures
#include <iostream.h>
#include <math.h>
struct complex
{
   int x;       // real part
   int y;       // imaginary part
   void read()
   {
      cout << "Real part ? ";
      cin >> x;
      cout << "Imaginary part ?";
      cin >> y;
   }
   void show( char *msg )
   {
      cout << msg << x;
      if( y < 0 )
         cout << "-i";
      else
         cout << "+i";
      cout << fabs(y) << endl;
   }
   void add( complex c2 )
   {
      x += c2.x;
      y += c2.y;
   }
};
void main()
{
   complex c1, c2, c3;
   cout << "Enter complex number c1 .." << endl;
   c1.read();
   cout << "Enter complex number c2 .." << endl;
   c2.read();
   c1.show( "c1 = " );
   c2.show( "c2 = " );
   c3 = c1;     // assignment
   c3.add( c2 );  // c3 = c3 + c2;
   c3.show( "c3 = c1 + c2 = " );
}
```

Run

```
Enter complex number c1 ..
Real part ? 1
Imaginary part ?2
Enter complex number c2 ..
Real part ? 3
Imaginary part ? 4
c1 = 1+i2
```

```
c2 = 3+i4
c3 = c1 + c2 = 4+i6
```

In `main()`, the statement

```
c1.read();
```

invokes the member function `read()`, defined in the structure `complex`. The data members of the variable `c1` are assigned with the input values. The statement,

```
c1.show( "c1 = " );
```

displays data members with suitable messages. The statement,

```
c3 = c1;    // assignment
```

assigns the contents of all the data members of the variable `c1` to corresponding members of `c2`. The statement,

```
c3.add( c2 );  // c3 = c3 + c2;
```

adds the contents of the variable `c2` to `c3`.

Note that, structures and classes in C++ exhibit the same set of features except that structure members are public by default, whereas class members are private by default. Most of the C++ programmers prefer to use a class to group data and functions; a structure to group only data which are logically related. Hence, through out this book, a construct called `class` (instead of `struct`) is used as a means for implementing OOP concepts. More details on classes can be found in the chapter: *Classes and Objects*.

8.11 Unions

A union allows the overlay of more than one variable in the same memory area. Normally, each and every variable is stored in a separate location and as a result, each one of these variables have their own addresses. Often, it is found that the variables used in a program appear only in a small portion of the source code. Consider the following situation to illustrate the benefits of union data type:

Suppose, a string of 200 bytes is needed to store *filename* in the first 500 lines of the code only, and another string of 400 bytes is needed to use as *buffer* in the rest of the code (that is from the 500th line onwards) Note that, no part of the code will access both the variables simultaneously. In such a situation, it would be a waste of memory if two arrays of 200 bytes and 400 bytes are defined; it requires 600 bytes of memory. The union provides a means by which the memory space can be shared, and only 400 bytes of memory is needed.

Declaring a Union

In terms of declaration syntax, the union is similar to a structure as shown in Figure 8.13. The method used to declare a structure is adopted to declare a union. A `union` data type is like a structure, except that it allows to define variables, which share storage space. Note the only change is the substitution of the keyword `struct` by the keyword `union`. The rest of the discussion regarding the declaration is the same as that given for the structure (i.e., even functions can be a part of union).

The compiler will allocate sufficient storage to accommodate the largest element in the union. Unlike a structure, members of a union variable occupy the same locations in memory (starting at the zero offsets). Thus, updating one member will *overwrite* the other. Elements of a `union` type variable are accessed in the same manner as the elements of a structure.

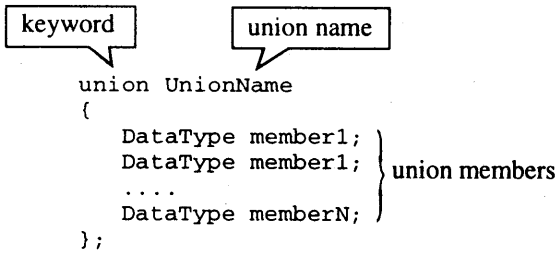


Figure 8.13: Union declaration

The memory space required for defining a variable of the union is:

```
max( sizeof(member1), sizeof( member2), ..., sizeof(memberN) )
```

That is, the member of biggest size should fit in the common memory space.

Defining Variables

Union variables can be defined at the point of union declaration or can be defined separately as and when required. Consider the following declaration:

```
union X                              // union declaration
{
   int a;
   char ch;
   double b;
};
```

The variables of the above union X can be defined as follows:

```
union X x1;
```

The storage space required to represent the variable x1 is max(sizeof(int), sizeof(char), sizeof(double)). At any point of time, the union variable can hold data of any one of its members. It is the responsibility of the programmer to decide to which of its members the data stored in the union variable is meaningful.

Member Access

Members of the union can be accessed using either the dot or the arrow (->) operator. It is similar to accessing the structure variable. Consider the following declaration:

```
union person
{
   char name[25];
   int idno;
   float salary;
};
```

The variables of the above union person can be defined as follows:

```
union person var1,*var2;  // var1 is value variable, var2 is pointer
```

The statement to assign the address of a variable var1 to the pointer variable var2 is as follows:

```
var2 = &var1;
```

The individual members can be accessed as follows:

```
var1.name                access the name
```

```
var1.idno      access the idno
var2->salary   access the salary
```

The members can be assigned in the same way as the members of a structure. For instance,

```
var1.idno = 20;
strcpy( var1.name, "Vijayashree" );
```

the content of the members of the union variable `var1` can be displayed as follows:

```
cout << var1.name;
```

The program `union.cpp` illustrates the usage of union to share the storage space.

```
// union.cpp: union of two strings
#include <iostream.h>
#include <string.h>
union Strings
{
   char filename[200];
   char output[400];
};
void main()
{
   Strings s;
   //.....
   strcpy( s.filename, "/cdacb/usr1/raj/oops/microkernel/pserver.cpp" );
   cout << "filename: " << s.filename << endl;
   //.....
   //.....
   strcpy(s.output,"OOPs is a most complex entity ever created by humans");
   cout << "output: " << s.output << endl;
   cout << "Size of union Strings = " << sizeof( Strings );
}
```

Run

```
filename: /cdacb/usr1/raj/oops/microkernel/pserver.cpp
output: OOPs is a most complex entity ever created by humans
Size of union Strings = 400
```

8.12 Differences between Structures and Unions

Structures and unions have the same syntax in terms of their declaration and definition of their variables. However, they differ in the amount of storage space required for their storage and the scope of the members.

Memory Allocation

The amount of memory required to store a structure variable is the sum of the size of all the members. On the other hand, in the case of unions, the amount of memory required is always equal to that required by its largest member. The program `sudiff.cpp` illustrates the memory requirements for variables of the structure and union types.

```
// sudiff.cpp: memory requirement for structures and unions
#include <iostream.h>
struct
{
   char name[25];
   int idno;
   float salary;
} emp;
union
{
   char name[25];
   int idno;
   float salary;
} desc;

void main()
{
   cout << "The size of the structure is " << sizeof(emp) << endl;
   cout << "The size of the union is " << sizeof(desc) << endl;
}
```

Run

```
The size of the structure is 31
The size of the union is 25
```

Operations on Members

Only one member of a union can be accessed at any given time. This is because, at any instant, only one of the union variables can be active. The general rule for determining the active member is: *only that member which is updated can be read.* At this point, the other variables will contain meaningless values. It is the responsibility of the programmer to keep track of the active members, The program `uaccess.cpp` illustrates accessing of a union variable and its members.

```
// uaccess.cpp: accessing of union members
#include <iostream.h>
#include <string.h>
union emp
{
   char name[25];
   int idno;
   float salary;
};
void show( union emp e )
{
   cout << "Employee Details ..." << endl;
   cout << "The name is " << e.name << endl;
   cout << "The idno is " << e.idno << endl;
   cout << "The salary is " << e.salary << endl;
}
```

```
void main()
{
   union emp e;    // or emp e;
   strcpy(e.name, "Rajkumar");
   show( e );
   e.idno = 10;
   show( e );
   e.salary = 9000;
   show( e );
}
```

Run

```
Employee Details ...
The name is Rajkumar
The idno is 24914
The salary is 2.83348e+26
Employee Details ...
The name is
The idno is 10
The salary is 2.82889e+26
Employee Details ...
The name is
The idno is -24576
The salary is 9000
```

The status of the variable e after execuion of each one of the following:

1. `strcpy(e.name, "Rajkumar");`
2. `e.idno = 10;` and
3. `e.salary = 9000;`

is shown in Figure 8.14a, 8.14b, 8.14c respectively. Note that, access of *non active* members will lead to meaningless values.

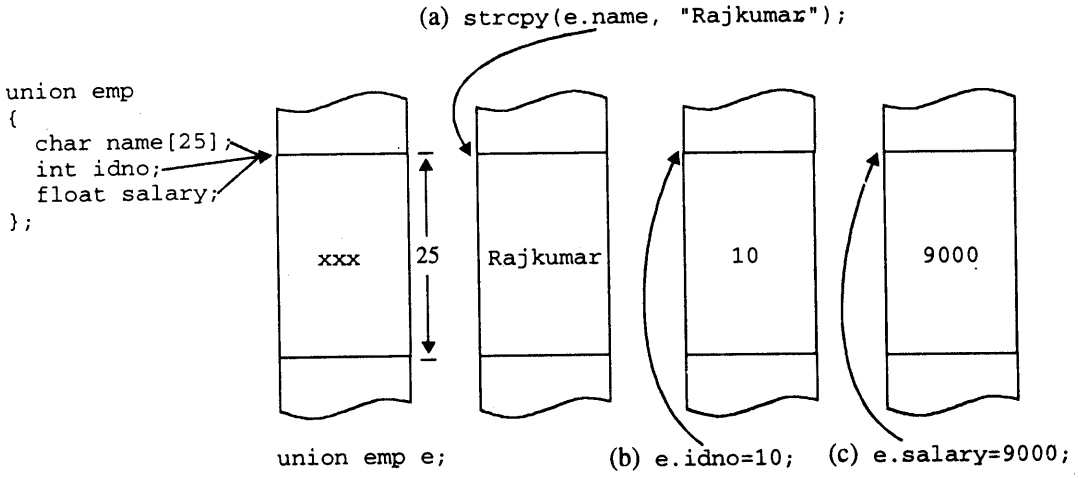


Figure 8.14: Union variable initialization

Operation on Unions

In addition to the features discussed above, the union has all the features provided by the structure except for minor changes, which is a consequence of the memory sharing capabilities of the union. This is made evident by the following legal operations.

- A union variable can be assigned to another union variable, if their tags are same.
- A union variable can be passed to a function as a parameter.
- The address of the union variable can be extracted by using the address-of operator (&). This union pointer can be passed to functions.
- A function can return a union or a pointer to the union.

Performing operations on the unions as a whole, for example, arithmetic or comparison operations are illegal.

Scope of a Union

The members of a union have the same scope as the union itself. It is illustrated in the program `uscope.cpp`. The union definition having no tag or instance variable is called *anonymous union*.

```
// uscope.cpp: scope of union declaration
#include <iostream.h>
void main()
{
   union          // anonymous union definition
   {
      int i;
      char c;
      float f;
   };
   i = 10;
   c = 9;
   f = 4.5;
   cout << "The value of i is " << i << endl;
   cout << "The value of c is " << c << endl;
   cout << "The value of f is " << f << endl;
}
```

Run

```
The value of i is 0
The value of c is
The value of f is 4.5
```

In the above program, the scope of the union definition is limited to `main()` and henoe, the scope of its members, `i`, `c` and `f` is limited to `main()`. In `main()`, they can be accessed like any other local variables. The only difference is that the variables share the same memory.

8.13 Bit-fields in Structures

C++ allows packing many data items into a single machine word for efficient and optimal usage of the storage space. This facility is useful when a program needs flags to keep track of status information related to various activities. Consider a program, which stores information about a person including the

following:

- Are you possessing any formal degree ?
- Are you employed ?
- Single or married ?
- Male or Female ?
- Are you a teenage ?
- Are you Indian ?

The simplest way of achieving the above task is to define six integer variables, each keeping the status of one item. This method requires `6*sizeof(int)` bytes of memory locations. Another mechanism of achieving it is through the use of bit masks (macros) as follows:

```
#define DEGREE     01
#define EMPLOYED   02
#define MARRIED    04
#define MALE       08
#define TEENAGE    16
#define INDIAN     32
```

Note that, the numbers must be powers of two, so that they can act as masks corresponding to the relevant bit positions, thus accessing the bits by shifting, masking, and complementing. For instance, the statement

```
flags |= DEGREE;
```

sets the first bit to 1 and the statement

```
flags &= ~MARRIED;
```

clears the second bit indicating that a person is unmarried. The conditional statement

```
if( flags & MARRIED )
  cout << "Married person";
else
  cout << "Unmarried person";
```

is valid. These idioms (mode of expressions) are easily prone to errors. As an alternative to this mechanism, C++ offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A *bit-field* or *field* in short, is a set of adjacent bits within a single implementation-defined storage unit called a *word*. The syntax of field definition and access is based on structures. For instance, the above `#define` statements could be replaced by the definition of six fields as follows:

```
struct
{
   unsigned int is_degree  : 1;
   unsigned int is_employed: 1;
   unsigned int is_married : 1;
   unsigned int is_male    : 1;
   unsigned int is_teenage : 1;
   unsigned int is_indian  : 1;
} flags;
```

It defines a variable called `flags` which contains six single-bit fields. The number following the colon represents the field width. The fields declared are of type `unsigned int` (can be `int`) to ensure that they are unsigned quantities.

The Individual fields are referenced in the same way as other structure members. For instance,

```
flags.is_married
```

expression accesses the contents of its corresponding bit. Fields act like integers and can be used in arithmetic expressions just like other integers. Thus, the previous examples can be written more naturally as follows:

```
flags.is_degree = 1;
```

sets the first bit to 1 and the statement

```
flags.is_married = 0;
```

clears the second bit, indicating that a person is unmarried. The conditional statement

```
if( flags.is_married )
  cout << "Married person";
else
  cout << "Unmarried person";
```

is valid.

Consider the following declaration which illustrates bit-fields of larger width:

```
struct with_bits
{
   unsigned first : 5;
   unsigned second : 9;
};
```

The identifier `with_bits` is a structure containing 2 members: `first` and `second`. The member `first` is an integer with 5 bits, and `second` is an integer with 9 bits. Both the numbers can be stored in a single 16-bit entity (even though they add up to 14 bits, a 14-bit entity cannot exist in memory), rather than two separate integers. It is illustrated in the program `share.cpp`.

```
// share.cpp: union and structure combined
#include <iostream.h>
struct with_bits
{
   unsigned first : 5;
   unsigned second : 9;
};
void main()
{
   union
   {
      with_bits b;
      int i;
   };
   i = 0;          // Both first and second are cleared to 0
   cout << "On i = 0: b.first = " << b.first << " b.second = "<  second;
   b.first = 9;    // first is set to 9; second remains 0
   cout << endl << "b.first = 9: ";
   cout << "b.first = " << b.first << " b.second = " << b.second;
}
```

Run

```
On i = 0: b.first = 0 b.second = 0
b.first = 9: b.first = 9 b.second = 0
```

In `main()`, the union defines two variables `b` and `i`, and they are stored in the same memory location. In a way, they can act as aliases. The statement,

```
i = 0;
```

clears the complete word and inturn clears members of the structure `with_bits`. The statement

```
b.first = 9;
```

updates only the first 5-bits of the word. Note: *the maximum size of each bit-field is sizeof(int).*

Review Questions

8.1 What are structures ? Justify their need with an illustrative example.

8.2 Why structures are called heterogeneous data-types ?

8.3 Explain storage organization of structure variables.

8.4 Write an interactive program, which processes date of birth using structures. Enhance the same supporting processing of multiple students date of birth.

8.5 Write a short note on passing structure type variables to a function, and suitability of different parameter passing schemes in different situations.

8.6 Develop a program for processing admission report. Use a structure which has elements representing information such as roll number, name, date of birth (nested structure), branch allotted. The functions processing members of a structure must be a part of a structure. The format of report is as follows:

```
-------------------------------------------------------------------
 Roll.no.          Name          Date of Birth     Branch Allotted
-------------------------------------------------------------------
 xx          xxxxxxxxxxxxxxx       dd/mm/yy           xxxxxxxx
-------------------------------------------------------------------
```

8.7 What are unions? Write a program to illustrate the use of the union.

8.8 What are the differences between structures and unions.

8.9 Write an interactive program to process complex numbers. It has to perform addition, subtraction, multiplication, and division of complex numbers. Print results in `x+iy` form.

8.10 Write a union declaration for representing register model of x86 family of microprocessors. Note that general purpose registers such as AX are also accessed by lower and higher word registers AH and AL respectively.

8.11 Consider the following structure declaration:

```
struct institution
{
   struct teacher {
       int empl_no;
       char name[20];
   };
   struct student  {
       int roll_no;
       char name[15];
   };
};
```

What is the `sizeof(institution)`, `sizeof(teacher)`, and `sizeof(student)`?

9

Pointers and Runtime Binding

9.1 Introduction

The use of pointers offers a high degree of flexibility in the management of data. Knowledge of memory organization plays a very important role for understanding the concept of pointers. As the name implies, pointer refers to the address identifying a programming element (data or function). Interestingly, the system main memory is organized into code and data area as shown in Figure 9.1. Although in many situations programming can be done without the use of pointers, their usage enhances the capability of the language to manipulate data. Dynamic memory allocation is a programming concept wherein the use of pointers becomes indispensable. For instance, to read the marks of a set of students and store them for processing, an array can be defined as follows:

```
float marks[100];
```

But this method limits the maximum number of students (to 100), which must be decided during the development of the program. On the other hand, by using dynamic allocation, the program can be designed so that the limit for the maximum number of students is restricted only by the amount of memory available in the system. The real power of C++ (of course C) lies in the proper use of pointers.

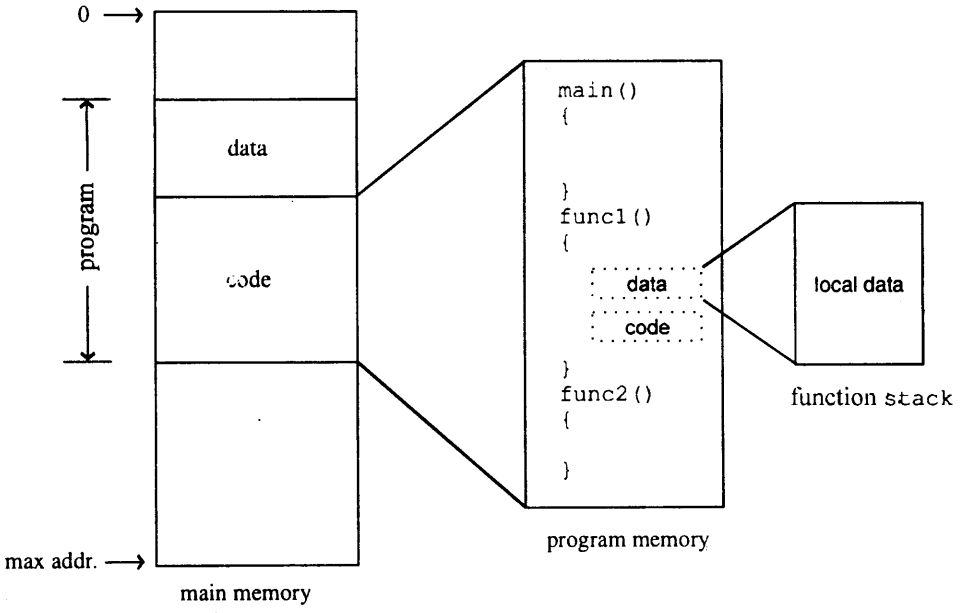


Figure 9.1: Primary memory organization

Memory is organized in the form of a sequence of byte-sized (8-bits per byte) locations or *storage cells* containing either program code or data. These bytes are numbered starting from zero onwards. The number associated with each cell (byte location) is known as its address or memory location. A pointer is an entity, which contains a memory address. In effect, a pointer is a number, which specifies a location in memory. The key concepts and terminology associated with memory organization are the following:

- Each byte in the memory is associated with a unique address.
- An address is a sequence of binary digits (0 or 1) of fixed length, used for labeling a byte in the memory.
- Address is a positive integer ranging from 0 to maximum addressing capability of the microprocessor (for instance, 8086 processor has 20-address lines and hence, it can address upto 2^{20} locations: 1 MB).
- Every element (data or program code) that is loaded into memory is associated with a valid range of addresses. i.e., each variable and function in the program starts at a particular location and spans across consecutive addresses from that point onwards depending upon the size of the data item.
- The number of bytes accessed by a pointer depends on the data type of an item to which it is a pointer.

The address stored in a pointer variable can be relative or absolute. Most of the modern systems use the relative addressing mode to access memory, by default. In relative addressing mode, an address consists of two components: the *base* (or the segment) and the *offset* address. The base or segment address designates a specific region of memory, and the offset specifies the distance of the desired memory location from the beginning of the segment. The effective address is computed by combining both the segment and offset values. In absolute mode, the address stored in a pointer is itself the effective address, and hence, memory can be directly accessed using this address. Note that, relative addressing requires mapping of logical address (offset) to physical address.

It is not always necessary to be aware of the segments and offsets while programming in C++, unless the pointer is used to hold the address of any device specific information. For instance, in IBM-PC and its compatibles, the display memory is located at the segment and offset value, `0xb800:0000`. (The display memory address changes from one video mode to another.)

9.2 Pointers and their Binding

Pointer is defined as a variable used to store memory addresses. It is similar to any other variable and has to be defined before using it, to hold an address. Just like, an integer variable can hold only integers, each pointer variable can hold only pointer to a specific data type such as `int`, `char`, `float`, `double`, etc., or any user defined data type).

The allocation of memory space for data structure (storage) during the course of program execution is called dynamic memory allocation. Dynamic variables so created can only be accessed with pointers. Thus, pointers offer tremendous flexibility in the creation of dynamic variables, accessing and manipulating the contents of memory location and releasing the memory occupied by the dynamic variables, which are no longer needed. (A more detailed account of dynamic memory allocation and de-allocation is discussed in the later sections of this chapter.) The usage of the pointer is essential in the following situations:

- Accessing array elements.
- Passing arguments to functions by address when modification of formal arguments are to be reflected on actual arguments.

- Passing arrays and strings to functions.
- Creating data structures such as linked lists, trees, graphs, etc.
- Obtaining memory from the system dynamically.

9.3 Address Operator &

All the variables defined in a program (including pointer variables) reside at specific addresses. It is possible to obtain the address of a program variable by using the address operator &(ampersand). When used as a prefix to the variable name, the & operator returns the address of that variable. The program `getaddr.cpp` illustrates the use of the & operator.

```
// getaddr.cppc: use of '&' operator to access address
#include <iostream.h>
void main()
{
   // define and initialize three integer variables
   int a = 100;
   int b = 200;
   int c = 300;
   // print the address and contents of the above variables
   cout << "Address " << &a << " contains value " << a << endl;
   cout << "Address " << &b << " contains value " << b << endl;
   cout << "Address " << &c << " contains value " << c << endl;
}
```

Run

```
Address 0xfff4 contains value 100
Address 0xfff2 contains value 200
Address 0xfff0 contains value 300
```

In `main()`, the statement

```
cout << "Address " << &a << ". contains value " << a << endl;
```

displays the address and contents of the variable `a`. The expression `&a` returns the address of the variable `a`. It should, however, be noted that the addresses printed by the above program, depend on the current configuration of a system. This is because the memory occupied by the program's variables depend on several factors such as memory management scheme, memory model, and the current status of the memory contents.

The output shows the addresses of the variables in hexadecimal notation, and they are in the decreasing order. From this, it is evident that *all automatic variables are created in the program's stack area and that the stack always grows from a higher to a lower memory address*. Further, each of the addresses differ from others by exactly two bytes, since integer variables are allocated 2 bytes of memory. The `sizeof()` operator can be used to determine the number of bytes allocated to each type of variable. The integer is the fundamental data type and hence its size depends on the processor word size, compiler, and operating system memory manager. For instance, the size of an integer data type in MS-DOS based machines is two bytes, whereas in UNIX based machines it is four bytes.

Sufficient care must be taken to avoid any kind of confusion between the following:

- unary address operator & which precedes a variable name.
- binary logical operator & which performs a bit-wise AND operation.

9.4 Pointer Variables

Pointers are also variables and hence, they must be defined in a program like any other variable. Rules for variable names and declaring pointers are the same as for any other data type. This naturally gives rise to questions about the data type of a pointer, size of memory allocated to a pointer and the format for defining different types of pointers.

Pointer Definition

When a pointer variable is defined, the C++ compiler needs to know the type of variable the pointer points to. The syntax of pointer variable definition is shown in Figure 9.2.

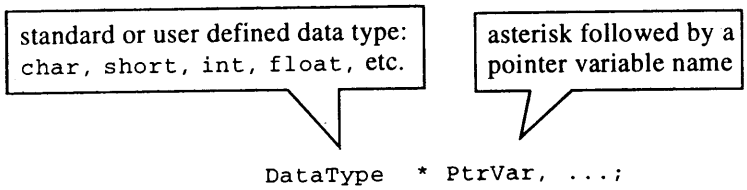


Figure 9.2: Syntax of pointer definition

`DataType` could be a primitive data type or user defined structure (such as structures and classes). The `PtrVar` could be any valid C++ variable name. The character star (`*`) following the `DataType` informs the compiler that the variable `PtrVar` is a pointer variable. The pointer so created can hold the address of any variable of the specified type. Some typical pointer definitions are:

```
int *int_ptr;   // int_ptr is a pointer to an integer
char *ch_ptr;   // ch_ptr is a pointer to a character
Date *d_ptr;    // d_ptr is a pointer to user defined data type
```

The pointer variable must be bound to a memory location. It can be achieved either by assigning the address of a variable, or by assigning the address of the memory allocated dynamically. The address of a variable can be assigned to a pointer variable as follows:

```
int_ptr = &marks;
```

where the variable `marks` is of type integer.

Pointer to characters (a string) can be defined as follows:

```
char *msg;
```

It can be initialized at the point of definition as follows:

```
char *msg = "abcd..xyz";
```

Or, it can also be initialized during execution as follows:

```
msg = "abcd..xyz";
```

Dereferencing of Pointers

Dereferencing is the process of accessing and manipulating data stored in the memory location pointed to by a pointer. The operator * (asterisk) is used to dereference pointers in addition to creating them. A pointer variable is dereferenced when the *unary operator* * (in this case, it is called as the *indirection operator)* is prefixed to the pointer variable or pointer expression. Any operation that is performed on the dereferenced pointer directly affects the value of the variable it points to. The syntax for *dereferencing pointers* is shown in Figure 9.3.

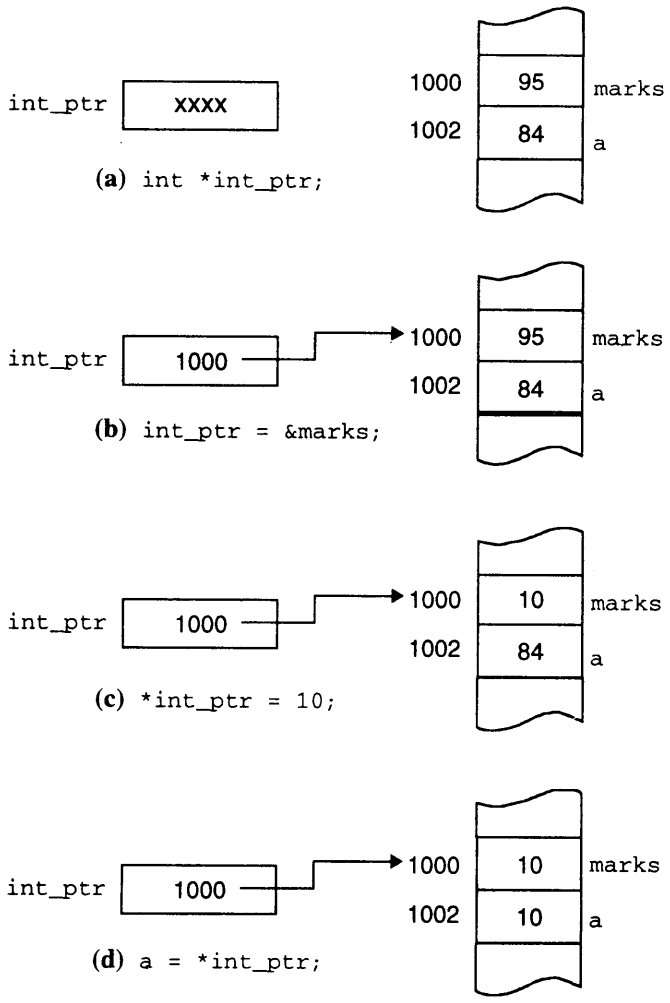


Figure 9.3: Pointers binding and dereferencing

Consider the statement

```
int_ptr = &marks;
```

It stores the address of the variable `marks` in the pointer variable `int_ptr`. The contents of the variable `marks` can be displayed using the following statement:

```
cout << *int_ptr;
```

Effectively, the above statement achieves the same result as the statement

```
cout << marks;
```

Thus, accessing information using pointers is called indirect addressing. It refers to accessing information, whose address is stored in a special type of variable, which is a pointer variable.

The contents of memory locations can be modified by using a pointer variable as follows:

```
*int_ptr = 25;
```

It assigns the value 25 to the memory location pointed to by the variable int_ptr. The contents of the memory location can be read by using the pointer variable as follows:

```
a = *int_ptr;
```

It assigns the contents of the memory location pointed to by the address stored in the variable int_ptr to the variable a of type integer. The program initptr.cpp illustrates the mechanism of pointer variable definition, binding and dereferencing.

```
// initptr.cpp: pointer (address variables) usage demonstration
#include <iostream.h>
void main ()
{
   int *iptr;        // pointer to integer, figure 9.4a
   int var1, var2;   // two integer variables, figure 9.4b
   var1 = 10;        // figure 9.4c
   var2 = 20;        // figure 9.4d
   iptr = &var1;     // figure 9.4e
   cout << "Address and contents of var1 is " << iptr << " and " << *iptr;
   iptr = &var2;     //  figure 9.4f
   cout<<"\nAddress and contents of var2 is " << iptr << " and " << *iptr;
   *iptr = 125;      // figure 9.4g
   var1 = *iptr + 1; // figure 9.4h
}
```

Run

```
Address and contents of var1 is 0x1f8afff4 and 10
Address and contents of var2 is 0x1f8afff2 and 20
```

In main(), the first statement

```
int *iptr;
```

specifies that iptr is a pointer to an integer. The asterisk prefixed to the variable name specifies that iptr is a pointer variable. The data type int specifies that iptr can point to any integer type item(s) stored in the main memory. The statement

```
int *iptr;        // pointer to integer, figure 9.4a
```

could also be written as

```
int* iptr;
```

It makes no difference as far as the compiler is concerned. But there are certain advantages in following the former convention (i.e., placing the * closer to the variable name). The compiler always associates the * with the pointer variable name rather than the data type, thus allowing both pointer variable type and non-pointer variable of a particular data type to be defined in a single definition. Thus, the following statements

```
int *iptr;        // pointer to integer, figure 9.4a
int var1, var2;   // two integer variables, figure 9.4b
```

are valid. They can also be written in a single equivalent statement as follows:

```
int *iptr, var1, var2;
```

An asterisk must be prefixed to the name of each pointer variable to define multiple pointers using a single statement. For instance, the statement,

```
float  *f1, *f2, *f3;
```

defines `f1`, `f2`, and `f3` as pointers to float variables.

The program `initptr.cpp` has highlighted the following important facts about pointers:

- The asterisk (*) used as an *indirection operator* has a different meaning from the asterisk used while defining pointer variables.
- Indirection allows the contents of a variable to be accessed and manipulated without using the name of the variable.

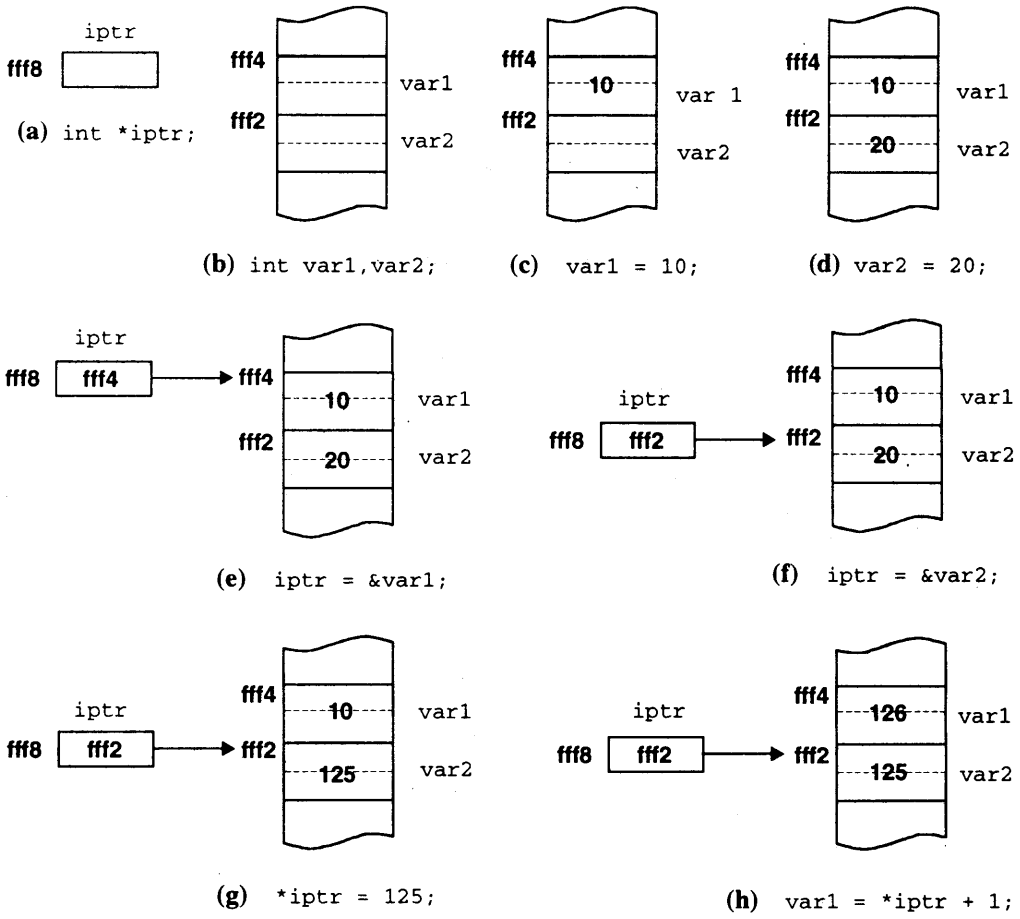


Figure 9.4: Dereferencing of pointers

All variables that can be accessed directly (by their names) can also be accessed indirectly by

means of pointers. The power of pointers becomes evident in situations, where indirect access is the only way to access variables in memory. Figure 9.4 gives a pictorial representation of accessing a variable using a pointer.

Pointers and Parameter Passing

Pointers provide a two way communication between the service requester and service provider. It is achieved by passing the address of the actual parameters instead of their contents. Any modification done to formal variables in the function will be automatically reflected in the actual parameters when they are passed by address. A program to swap two numbers is listed in swap.cpp.

```
// swap.cpp: swap 2 numbers using pointers
#include <iostream.h>
void swap(float *, float *);
void main()
{
   float a, b;
   cout << "Enter real number <a>: ";
   cin >> a;
   cout << "Enter real number <b>: ";
   cin >> b;
   // Pass address of the variables whose values are to be swapped
   swap(&a, &b);   // figure 9.6a
   cout << "After swapping ......\n";
   cout << "a contains " << a << endl;
   cout << "b contains " << b;
}
void swap( float *pa, float *pb ) // function to swap two numbers
{
   float temp;
   temp = *pa;   // figure 9.6b
   *pa  = *pb;   // figure 9.6c
   *pb  = temp;  // figure 9.6d
}
```

Run

```
Enter real number <a>: 10.5
Enter real number <b>: 20.9
After swapping ......
a contains 20.9
b contains 10.5
```

In main(), the statement

```
swap(&a, &b);
```

assigns addresses of the actual parameters to the formal parameters, which are of type pointers. However, they are manipulated differently (see Figure 9.5). In main(), the parameters are accessed directly with their names whereas in swap(), they are accessed using the *indirection operator.*

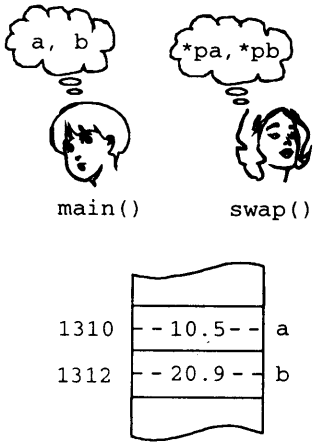


Figure 9.5: Data addressing in different perspectives

In `swap()`, accessing contents of the memory location pointed to by the variable `pa`, actually accesses the contents of the variable `a`. Similarly, accessing the contents of the memory location pointed to by the variable `pb` actually access the contents of the variable `b`. Hence, swapping the contents of memory using pointer variables `pa` and `pb` along with the indirection operator will in fact exchange the contents of the actual parameters `a` and `b` (passed by caller) as shown in Figure 9.6.

9.5 Void Pointers

Pointers defined to be of a specific data type cannot hold the address of some other type of variable i.e., it is syntactically incorrect in C++ to assign the address of (say) an integer variable to a pointer of type `float`. Consider the following definitions

```
float *f_ptr; // pointer to float
int   my_int; // integer variable
```

The assignment of incompatible variable address to a pointer variable in a statement such as

```
f_ptr = &my_int;
```

results in compilation error. Such type-compatibility problems can be overcome by using a general-purpose pointer type called *void pointer*. The format for declaring a *void pointer* is as follows:

```
void *v_ptr;  // define a pointer to void
```

It uses the reserved word `void` for specifying the type of the pointer. Pointers defined in this manner do not have any type associated with them and can hold the address of any type of variable. The following are some valid C++ statements:

```
void *vd_ptr;
int  *it_ptr;
int  invar;
char chvar;
float flvar;
vd_ptr = &invar;  // valid
vd_ptr = &chvar;  // valid
```

```
vd_ptr = &flvar;  // valid
it_ptr = &invar;  // valid
```

The following are some invalid statements:

```
it_ptr = &chvar;  // invalid
it_ptr = &flvar;  // invalid
```

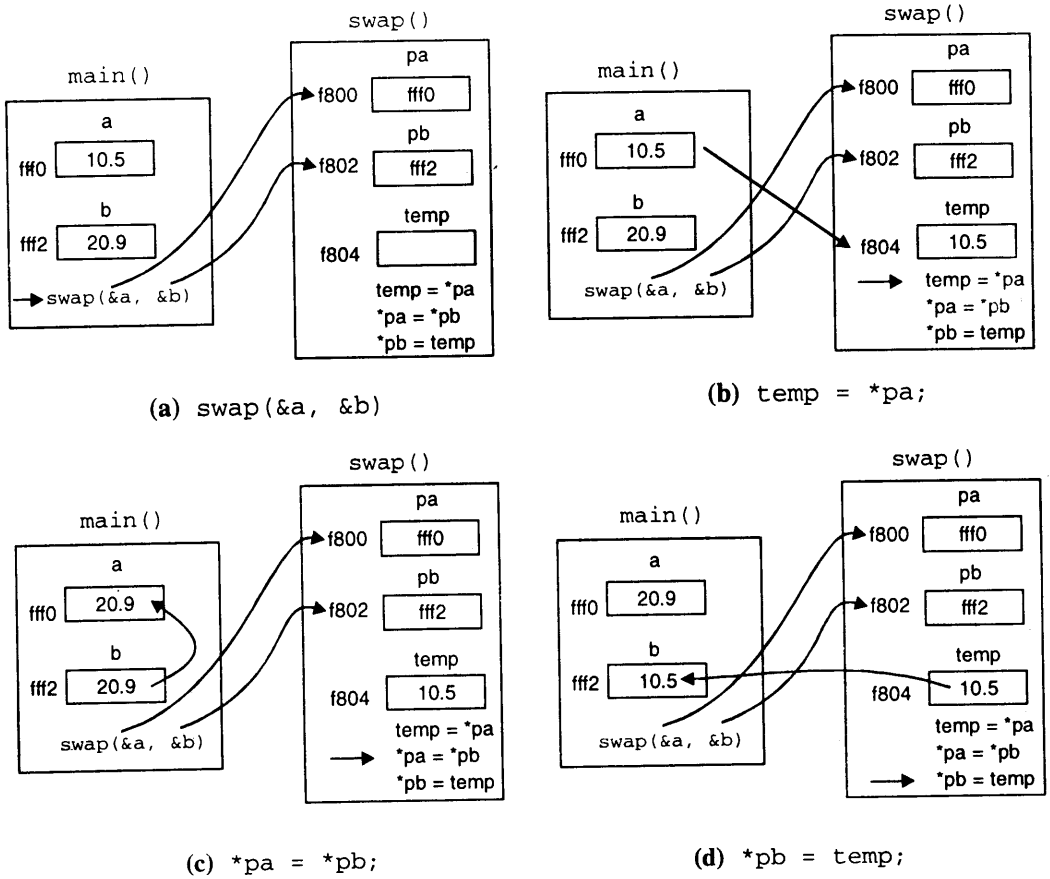


Figure 9.6: Swapping of two numbers.

Pointers to `void` cannot be directly dereferenced like other pointer variables using the *indirection operator*. Prior to dereferencing a pointer to `void`, it must be suitably typecasted to the required data type. The program `voidptr.cpp` illustrates the typecasting of void pointers while accessing memory locations pointed to by them.

```
// voidptr.cpp: the use of void pointers to hold pointer of any type
#include <iostream.h>
void main()
{
   int i1 = 100;      // define and initialize int i1 to 100
```

```
   float f1 = 200.5; // define and initialize float f1 to 200.50
   void *vptr;       // define pointer to void
   vptr = &i1;       // pointer assignment
   cout << "i1 contains " << *((int *) vptr) << endl;
   vptr = &f1;       // pointer assignment
   cout << "f1 contains " << *((float *) vptr);
}
```

Run

```
i1 contains 100
f1 contains 200.5
```

The expression `*((float*)vptr)` in the statement

```
cout << "f1 contains " << *((float *) vptr);
```

displays the contents of the variable `f1` using a `void` pointer variable with typecasting. Figure 9.7 indicates various components of the expression `*((float*)vptr))`. When a function is designed to do similar operations on different data types, `void` pointers can be used to pass parameters to the function.

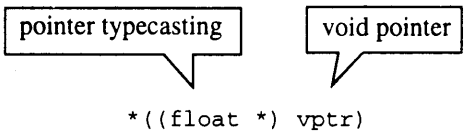


Figure 9.7: Typecasting void pointer

9.6 Pointer Arithmetic

The size of the data type to which the pointer variable refers is the number of bytes of memory accessed when the pointer variable is dereferenced using the indirection operator. The number of bytes accessed by using a pointer depends on its type, but the size of the pointer variable remains the same irrespective of the data type to which it is pointing (see Table 9.1). The size of the pointer variable is large enough to hold the memory address. For example, when dereferenced (in a particular implementation of the C++ compiler—on 16-bit system),

- a pointer to an integer accesses 2 bytes of memory
- a pointer to a char accesses 1 byte of memory
- a pointer to a float accesses 4 bytes of memory
- a pointer to a double accesses 8 bytes of memory

The C++ language allows arithmetic operations to be performed on pointer variables. It is, however, the responsibility of the programmer to see that the result obtained by performing pointer arithmetic is the address of relevant and meaningful data.

The arithmetic operators available for use with pointers can be classified as

- Unary operators : ++ (increment) and -- (decrement).
- Binary operators : + (addition) and - (subtraction).

Data type	Data size	Pointer type	
		near	far
char	1	2	4
short	2	2	4
int	2 (16-bit compiler) 4 (32-bit compiler)	2	4
long	4	2	4
float	4	2	4
double	8	2	4

Table 9.1: Size of data types and their pointers

The following are some of the examples of pointer arithmetic:

```
int a, b, *p, *q;
p = -q;                    // Illegal use of pointer
p <<= 1;                   // Illegal use of pointer
p = p - b;                 // Valid
p = p - q;                 // Invalid: Nonportable pointer conversion
p = (int *)(p - q);        // Valid
p = p - q - a;             // Invalid: Nonportable pointer conversion
p = (int *)(p - q) - a;    // Valid
p = p + a;                 // Valid
p = p + q;                 // Invalid pointer addition
p = p + q + a;             // Invalid pointer addition
p = p * q;                 // Illegal use of pointer
p = p * a;                 // Illegal use of pointer
p = p / q;                 // Illegal use of pointer
p = p / b;                 // Illegal use of pointer
p = a / p;                 // Illegal use of pointer
a = *p ** q;               // Valid and it is same as a = (*p) * (*q);
```

The C++ compiler takes into account the size of the data type being pointed, while performing arithmetic operations on a pointer. For example, if a pointer to an integer is incremented using the ++ operator (preceding or succeeding the pointer), then the initial address contained in the pointer is incremented by two and not one, assuming that an integer occupies two bytes in memory. Similarly, incrementing a pointer to float causes the initial address contained in the float pointer to be actually incremented by 4 and not 1 (if the size of the float variable is 4 on the machine). In general, a pointer to some type, d_type (where d_type can be primitive or user defined data type), when incremented by an integral value i, has the following effect:

```
(current address in pointer) + i * sizeof(d_type)
```

Consider the following statements:

```
float *sum;
char *name;
```

A statement such as

```
sum++; or ++sum;
```

advances the pointer variable sum to point to the next element. If the pointer variable sum holds the address 1000, on execution of the above statement, the variable sum will hold the address (1000+4) = 1004 since the size of float is 4 bytes. Similarly, when a statement such as

```
name++; or ++name;
```

is executed, and if the pointer variable name points to address 2000 earlier, then it will hold the address (2000+1), since the size of char is one byte. This concept applies to all arithmetic operations performed on pointer variables.

When a pointer variable is incremented, its value actually gets incremented by the size of the type to which it points. For example, let pi be a pointer to an integer defined with the statement

```
int* pi;
```

Also, let pi point to the memory location 1020. i.e., the number 1020 is stored in the pointer pi. Now, a statement which increments pi, such as

```
pi++;
```

will add two to pi, making it 1022 (assuming that the size of an integer is 2 bytes). This makes pi point to the next integer. Similarly, the statement

```
pi--;
```

will decrement the value of pi by 2. The pointer arithmetic on different types is shown in Table 9.2.

Pointer variable	Pointer value	Pointer increment	Pointer value after increment
char * a;	10	a++;/++a; a=a+3;	11(a+sizeof (char)) 13(a+sizeof (char)*3)
int * b;	10	b++;/++b; b=b+2;	12(b+sizeof (int)) 14(b+sizeof (int)*2)
long * c;	10	c++;/++c; c=c+3;	14(c+sizeof (long)) 22(c+sizeof (long)*3)
float * d;	10	d++;/++d; d=d+2;	14(d+sizeof (float)) 18(d+sizeof (float)*2)
double * e;	10	e++;/++e; e=e+2;	18(e+sizeof (double)) 26(e+sizeof(double)*2)

Table 9.2: Pointer arithmetic

Pointer arithmetic becomes significant for accessing and processing array elements efficiently (a more detailed account of array processing with pointers is taken up later in this chapter). Note that

pointer arithmetic cannot be performed on void pointers without typecasting, since they have no type associated with them.

The elements of an array can be efficiently accessed by using a pointer. The program `ptrarr1.cpp` illustrates the use of pointer holding the address of arrays and pointer arithmetic in manipulating large amount of data stored in sequence.

```
// ptrarr1.cpp: smallest in an array of 'n' elements using pointers
#include <iostream.h>
void main()
{
   int i,n, small, *ptr, a[50];
   cout  << "Size of the array ? ";
   cin >> n;
   cout << "Array elements ?\n";
   for (i = 0; i < n; i++)
      cin >> a[i];
   //  assign address of a[0] to pointer 'ptr'. This can be done in two
   // way: 1. ptr = &a[0];  2. ptr = a;
   ptr = a;
   // contents of a[0] assigned to small
   small = *ptr;
   // pointer points to next element in the array i.e., a[1]
   ptr++;
   // loop n-1 times to search for smallest element in the array
   for (i = 1; i < n; i++)
   {
      if(small > *ptr)
         small = *ptr;
      ptr++;   // pointer is incremented to point to a[i+1]
   }
   cout << "Smallest element is " << small;
}
```

Run

```
Size of the array ? 5
Array elements ?
4 2 6 1 9
Smallest element is 1
```

In `main()`, the statement

```
ptr = a;
```

assigns the address of the 0^{th} element of the array to the integer pointer `ptr`. Hence, the statement

```
small = *ptr;
```

effectively assigns the value of `a[0]` to the variable `small`. When `ptr` is incremented, the value stored in `ptr` is incremented by `sizeof(int)` (i.e., = 2 in DOS and = 4 in UNIX) to point to the next element of the array.

It is interesting to note that the name of the array represents the starting address of the array i.e., it is the address of the first element in the array. Hence, the expression `a[i]` can also be represented by the expression `*(a+i)`.

9.7 Runtime Memory Management

C++ provides two special operators new and delete to perform memory allocation and deallocation at runtime respectively. These operators with their syntax and suitable examples are already discussed in the earlier chapter on *Moving from C to C++*. An additional discussion on new operator follows:

The new operator must always be supplied with a data type in place of type-name. Items surrounded by angle brackets are optional. The syntax of new operator is as follows:

```
<::> new <new-args> type-name <(initializer)>
<::> new <new-args> (type-name) <(initializer)>
```

The components present in the syntax has the following meaning:

- `::` operator, invokes the global version of new.
- new-args can be used to supply additional arguments to new. It is used when the program has an overloaded version of new that matches the optional arguments.
- initializer, if present, is used to initialize the memory.

A request for non-array allocation uses the appropriate operator `new()` function. Any request for array allocation will call the appropriate operator `new[]()` function. Selection of the operator is done as follows:

- By default, the operator `new[]()` calls the operator `new()`
- If a class Type has an overloaded version of operator `new[]()`, arrays of Type will be allocated using `Type::operator new[]()`
- If a class Type has an overloaded version of new and it is not the array allocation operator `new[]()`, then the arrays of Type will be allocated using `Type::operator new()`
- If none of the above cases apply, the `global ::operator new()` is used.

More details on dynamic objects is discussed in later chapters.

Handling Errors for the new Operator

The new operator offers dynamic storage allocation similar to the standard library function `malloc`. It is particularly designed keeping OOPs in mind and throws an exception if the allocation fails. For more details on handling exceptions raised by the new operator, refer to the chapter on *Exception Handling*.

The user can define a function to be invoked when the new operator fails. The new operator can be informed about the new-handler function, by using `set_new_handler()` and pass a pointer to the new-handler. The new operator can be configured to return NULL on failure as follows:

```
set_new_handler(0).
```

It sets the handler to NULL so that the new operator returns NULL when it fails to allocate the requested amount of memory and thus exhibiting the behavior of the standard function `malloc()`. The program newhand.cpp illustrates the mechanism of handling the failure of memory allocation.

```
// newhand.cpp: new operator memory allocation test
#include <iostream.h>
#include <process.h>
#include <new.h>
void main(void)
{
```

```
   int * data;
   int size;
   set_new_handler( 0 );
   cout << "How many bytes to allocate: ";
   cin >> size;
   if( (data = new int[ size ]) )
      cout << "Memory allocation success, address = " << data;
   else
   {
      cout << "Could not allocate. Bye ...";
      exit(1);
   }
   delete data;
}
```

Run1

```
How many bytes to allocate: 100
Memory allocation success, address = 0x16be
```

Run2

```
How many bytes to allocate: 30000
Could not allocate. Bye ...
```

Note: A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

9.8 Pointers to Pointers

C++ allows programmers to define a pointer to pointers, which offers flexibility in handling arrays, passing pointer variables to functions, etc. The syntax for defining a pointer to pointer is:

```
DataType  **PtrToPtr;
```

which uses two * symbols (placed one beside the other). It implies that `PtrToPtr` is a pointer-to-a-pointer addressing a data object of type `DataType`. This feature is often used for representing two dimensional arrays. The program `ptr2ptr.cpp` illustrates the format for defining and using a pointer to another pointer.

```
// ptr2ptr.cpp: definition and use of pointers to pointers
#include <iostream.h>
void main(void)
{
   int *iptr;          // iptr as a pointer to an integer, figure 9.8a
   int **ptriptr;      // Defines pointer to int pointer, figure 9.8b
   int data;           // Some integer location, figure 9.8c
   iptr = &data;       // iptr now points to data, figure 9.8d
   ptriptr = &iptr;    // ptriptr points to iptr, figure 9.8e
   *iptr = 100;        // Same as data = 100, figure 9.8f
   cout << "The variable 'data' contains " << data << endl;
   **ptriptr = 200;    // Same as data = 200, figure 9.8g
   cout << "The variable 'data' contains " << data << endl;
   data = 300;         // figure 9.8h
```

```
   cout << "ptriptr is pointing to " << **ptriptr << endl;
}
```

Run

```
The variable 'data' contains 100
The variable 'data' contains 200
ptriptr is pointing to 300
```

In `main()`, the statement

```
int **ptriptr;
```

creates a pointer variable which holds a pointer to another pointer variable. The statement

```
ptriptr = &iptr;
```

assigns address of the pointer variable `iptr` to `ptriptr`. The value pointed by `iptr` can also be accessed by `ptriptr` as follows:

```
**ptriptr
```

The expression `**ptriptr` effectively accesses the contents of the variable data. The various operations on the pointer to a pointer are shown in Figure 9.8.

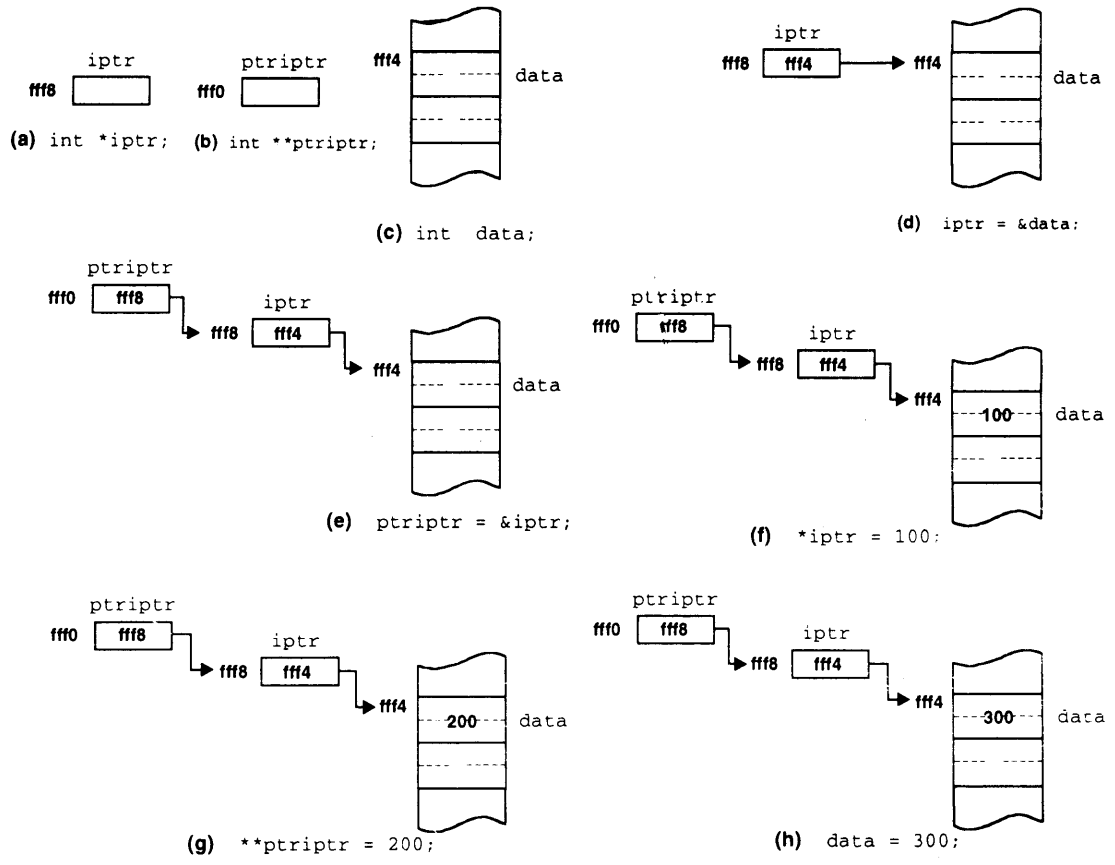


Figure 9.8: Pointers to pointer and dereferencing

Passing Address of a Pointer

When a pointer variable is defined, a memory location for the pointer is allocated, but it will not be initialized. Before using the pointer variable, it should be initialized. If the pointer variable has to be initialized in a function other than where it is defined, then the pointer's address has to be passed to the function. The contents of the *pointer to a pointer* variable can be used to access or modify the pointer type formal variable. The program `big.cpp` illustrates passing the address of a pointer, so that the pointer can be made to point to a desired variable (in this program, it is the biggest of two integers).

big.cpp: program to find the biggest number using pointers

```
#include <iostream.h>
void FindBig(int *pa, int *pb, int **pbig)
{
   // compare the contents of *pa and *pb and assign their address to pbig
   if( *pa > *pb )
      *pbig = pa;
   else
      *pbig = pb;
}
void main()
{
   int a, b, *big;
   cout << "Enter two integers: ";
   cin >> a >> b;
   FindBig( &a, &b, &big );
   cout << "The value as obtained from the pointer: " << *big;
}
```

Run

```
Enter two integers: 10 20
The value as obtained from the pointer: 20
```

In `main()`, the statement

```
FindBig( &a, &b, &big );
```

passes `a`, `b`, and `big` variables by address. It assigns the address of the variable `a` or `b` to the pointer variable `big`. In `FindBig()`, the statement

```
*pbig = pa;
```

effectively stores the address of the variable `a` in the pointer variable `big`, which is defined in the `main()` function.

9.9 Array of Pointers

An array of pointers is similar to an array of any predefined data type. As a pointer variable always contains an address, an array of pointers is a collection of addresses. These can be addresses of ordinary isolated variables or addresses of array elements. The elements of an array of pointers are stored in the memory just like the elements of any other kind of array. All rules that apply to other arrays also apply to the array of pointers.

The syntax for defining an array of pointers is the same as array definition, except that the array name is preceded by the star symbol during definition as follows:

DataType **ArrayName*[*ARRSIZE*];

An array of pointers is useful for holding a pointer to a list of strings. They can be utilized in implementing algorithms involving excessive data movements. It is a traditional style to sort data, by data movement. This method of sorting incurs much overhead in terms of both the time and space complexity, as it requires temporary space for exchanging the data between the records and has excessive data movement. This is especially true if the size of the data being sorted is large. Pointers can be utilized to perform the same with much flexibility and less overhead. In this method, instead of data exchange, pointers are exchanged to accomplish the same task. The program `sortptr.c` illustrates a method of sorting data without swapping their contents.

```
// sortptr.cpp: sorting of strings by pointer movement
#include <iostream.h>
#include <string.h>
// bubble sort algorithm based sorting function. It speeds up sorting
// by exchanging the pointers instead of heavy data movement
void SortByPtrExchange( char ** person, int n )
{
   int i, j, flag;
   char *temp;
   for( i = 0; i < n-1; i++ )    // for i = 0 to n-2
   {
      flag = 1;
      for( j = 0; j < (n-1-i); j++ )   // for j = 0 to (n-i-2)
      {
         if( strcmp( person[j], person[j+1] ) > 0 )
         {
            flag = 0;   // still not sorted and requires next iteration
            // exchange pointers
            temp = person[j];
            person[j] = person[j+1];
            person[j+1] = temp;
         }
      }
      if( flag )
         break; // data are in sorted order now; no need of next iteration
   }
}
void main()
{
   int i, n = 0;
   char *person[100];
   char choice;
   do
   {
      person[n] = new char[40];  // allocate space for a string
      cout << "Enter Name: ";
      cin >> person[n++];
      cout << "Enter another (y/n) ? ";
      cin >> choice;
   } while( choice == 'y' );
```

```
   cout << "Unsorted list: ";
   for( i = 0; i < n; i ++ )
      cout << endl << person[i];
   SortByPtrExchange( person, n );
   cout << endl << "Sorted list: ";
   for( i = 0; i < n; i ++ )
      cout << endl << person[i];
   // release memory allocated
   for( i = 0; i < n; i++ )
      delete person[i];
}
```

Run

```
Enter Name: Tejaswi
Enter another (y/n) ? y
Enter Name: Prasad
Enter another (y/n) ? y
Enter Name: Prakash
Enter another (y/n) ? y
Enter Name: Sudeep
Enter another (y/n) ? y
Enter Name: Anand
Enter another (y/n) ? n
Unsorted list:
Tejaswi
Prasad
Prakash
Sudeep
Anand
Sorted list:
Anand
Prakash
Prasad
Sudeep
Tejaswi
```

In `main()`, the statement

```
person[n] = new char[40];
```

allocates 40 bytes of memory to the $(n+1)^{th}$ element and stores its memory address in the array of pointers to strings indexed by n. The statement

```
SortByPtrExchange( person, n );
```

invokes the sorting function by passing the array of pointers and *data count* as actual parameters. Note that, array is passed to a function just by mentioning its name. This is equivalent to passing an entire array; the address of the first element of an array can be used to access any element in the array by using offset values. The data sorted by `SortByPtrExchange()` do not change their physical location (see Figure 9.9). The effect of sorting is seen when strings are accessed using pointers in a sequence.

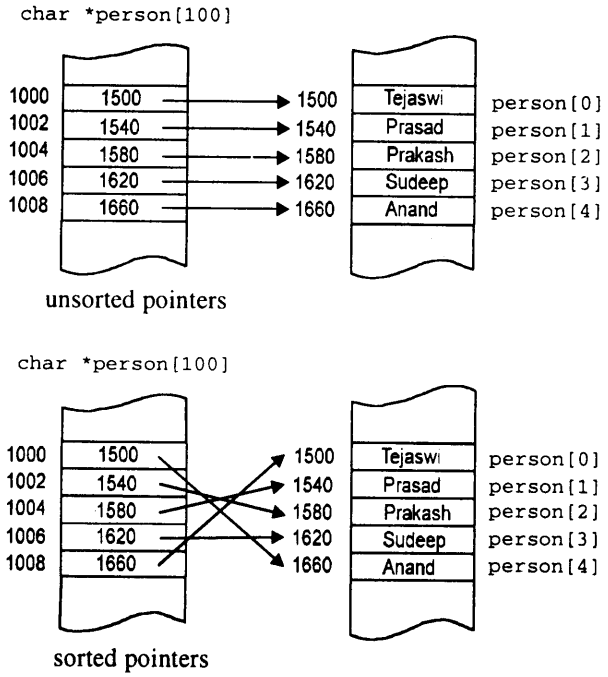


Figure 9.9: Sorting using pointers

Precedence of * and [ ] Operators

In C++, the notations `*p[3]` and `(*p)[3]` are different since * operator has a lower precedence than [ ] operator. The following examples illustrate the difference between these two notations:

1. `int *data[10];`

   It defines an array of 10 pointers. The increment operation such as

   `data++; or ++data;`

is invalid; the array variable `data` is a constant pointer.

2. `int (*data)[10];`

   It defines a pointer to an array of 10 elements. The increment operation such as

   `data++; or ++data;`

is invalid; the variable `data` will point beyond 10 integers, i.e., `10 *sizeof(int)` will be added to the variable `data`. The program `show.cpp` illustrates the use of defining a pointer to a matrix having arbitrary number of rows and fixed number of columns.

```
//show.cpp: matrix of unknown number of rows and known number of columns
#include <iostream.h>
void show( int a[][3], int m )
{
   int (*c)[3];   // pointer to an array of 3 elements
   c = a;
   for( int i = 0; i < m; i++ )
   {
```

```
      for( int j = 0; j < 3; j++ )
         cout << c[i][j] << " ";
      cout << endl;
   }
}
void main()
{
   int c[2][3]={{1,2,3}, {4,5,6}};
   show(c, 2);
}
```

Run

```
1 2 3
4 5 6
```

In `show()`, the statement

```
int (*c)[3];
```

defines a pointer to an array of three elements. It is useful for processing two dimensional array parameter declared with unknown number of rows. The statement

```
c = a;
```

assigns the address of a two dimensional array having three columns. The variable `c` allows to access all the array elements in the same way as a matrix. It allows pointer increment operations such as

```
c++; or ++c;
```

It increments pointer by `3*sizeof(int)`.

9.10 Dynamic Multi-dimensional Arrays

Pointers permit the creation of multi-dimensional arrays dynamically so that the amount of memory required by the array can be determined at runtime depending on the problem size. A two dimensional array can be thought of as a collection of a number of one dimensional arrays each representing a row. The 2D array is stored in memory in the row major form and it can be created dynamically using the following steps:

1. Define a pointer to pointers matrix variable: `int **p;`
2. Allocate memory for storing pointers to all rows of a matrix:

```
p = new int *[ row ];
```

3. Allocate memory for all column elements:

```
for( int i = 0; i < row; i++ )
    p[i] = new int[ col ];
```

The model of a dynamic matrix is shown in Figure 9.10. It is possible to access the two dimensional array elements using pointers in the same way as the one-dimensional array. Each row of the two dimensional array is treated as one dimensional array. The name of the array indicates the starting address of the array. The expressions `arrayname[i]` and `(arrayname+i)` point to the i^{th} row of the array. Therefore, `*(arrayname+i)+j` points to the j^{th} element in the i^{th} row of the array. The subscript `j` actually acts as an offset to the base address of the i^{th} row. The two dimensional dynamic matrix elements can also be accessed by using the notation `a[i][j]`.

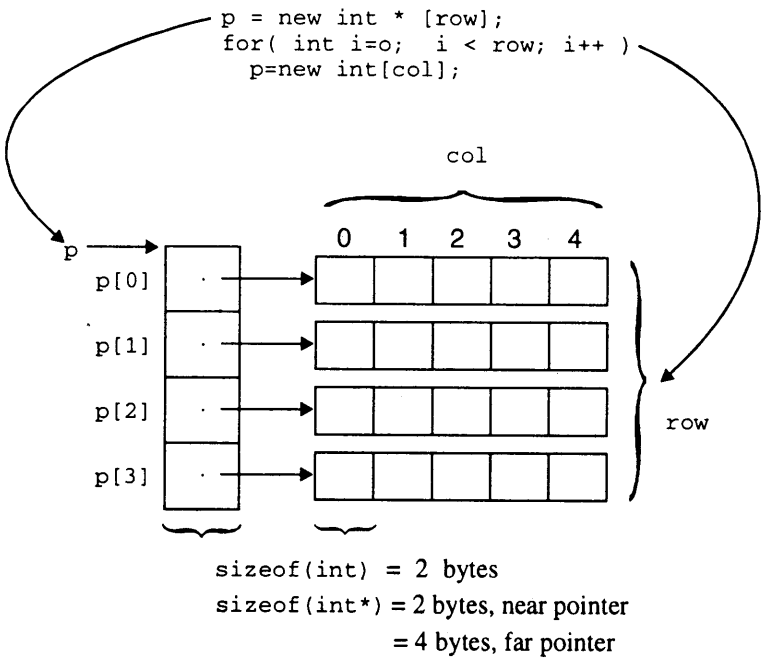


Figure 9.10: Model of dynamic matrix

```
// matrix.cpp: matrix manipulation and dynamically resource allocation
#include <iostream.h>
#include <process.h>
int **MatAlloc( int row, int col )
{
   int **p;
   p = new int *[ row ];
   for( int i = 0; i < row; i++ )
      p[i] = new int[ col ];
   return p;
}
void MatRelease( int **p, int row )
{
   for( int i = 0; i < row; i++ )
      delete p[i];
   delete p;
}
void MatRead(int **a, int row, int col )
{
   int i, j;
   for( i = 0; i < row; i++ )
      for( j = 0; j < col; j++ )
      {
         cout << "Matrix[" << i << "," << j << "] = ? ";
```

```
            cin >> a[i][j];
        }
}
// multiplication of matrices, c3.mul(c1, c2): c3 = c1*c2
void MatMul( int **a, int m, int n, int **b, int p, int q, int **c )
{
   int i, j, k;
   if( n != p )
   {
      cout << "Error: Invalid matrix order for multiplication";
      exit( 1 );
    }
    for( i = 0; i < m; i++ )
      for( j = 0; j < q; j++ )
      {
         c[i][j] = 0;
         for( k = 0; k < n; k++ )
            c[i][j] += a[i][k] * b[k][j];
      }
}
void MatShow( int **a, int row, int col )
{
   int i, j;
   for( i = 0; i < row; i++ )
   {
      cout << endl;
      for( j = 0; j < col; j++ )
         cout << a[i][j] << " ";
   }
}
void main()
{
   int **a, **b, **c;
   int m, n, p, q;
   cout << "Enter Matrix A details..." << endl;
   cout << "How many rows ? ";
   cin >> m;
   cout << "How many columns ? ";
   cin >> n;
   a = MatAlloc( m, n );
   MatRead( a, m, n );
   cout << "Enter Matrix B details..." << endl;
   cout << "How many rows ? ";
   cin >> p;
   cout << "How many columns ? ";
   cin >> q;
   b = MatAlloc( p, q );
   MatRead( b, p, q );
   c = MatAlloc( m, q );
   MatMul( a, m, n, b, p, q, c );
```

```
   cout << "Matrix C = A * B ...";
   MatShow( c, m, q );
}
```

Run

```
Enter Matrix A details...
How many rows ? 3
How many columns ? 2
Matrix[0,0] = ? 1
Matrix[0,1] = ? 1
Matrix[1,0] = ? 1
Matrix[1,1] = ? 1
Matrix[2,0] = ? 1
Matrix[2,1] = ? 1
Enter Matrix B details...
How many rows ? 2
How many columns ? 3
Matrix[0,0] = ? 1
Matrix[0,1] = ? 1
Matrix[0,2] = ? 1
Matrix[1,0] = ? 1
Matrix[1,1] = ? 1
Matrix[1,2] = ? 1
Matrix C = A * B ...
2 2 2
2 2 2
2 2 2
```

Three-dimensional Array

A three dimensional array can be thought of as an array of two dimensional arrays. Each element of a three dimensional array is accessed using three subscripts, one for each dimension.

As usual, the array name points to the base address of the three dimensional array. The array name with a single subscript `i` contains the base address of the i^{th} two-dimensional array. Hence `arrayname[i]` or `(arrayname+i)` is the address of the i^{th} two dimensional array. The expression `arrayname[i][j]` or `*(arrayname+i)+j` represents the base address of the j^{th} row in the i^{th} two dimensional array. Similarly, the expression `*(*(arrayname+j)+k)` points to the k^{th} element in the j^{th} row in the i^{th} two dimensional array. The program `3ptr.cpp` illustrates these concepts.

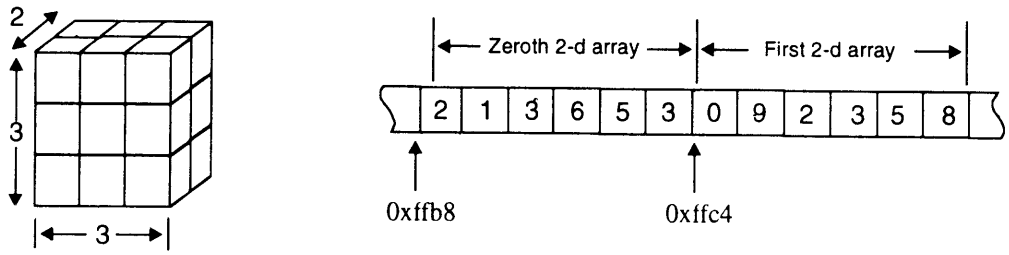
```
// 3ptr.cpp: pointer to 3-dimensional arrays
#include <iostream.h>
void main()
{
   int arr[2][3][2] ={ {{2,1},{3,6},{5,3}}, {{0,9},{2,3},{5,8}}};
   cout << arr << endl;
   cout << *arr << endl;
   cout << **arr << endl;
   cout << ***arr << endl;
```

```
  cout << arr+1 << endl;
  cout << *arr+1 << endl;
  cout << **arr+1 << endl;
  cout << ***arr+1 << endl;
  for( int i=0; i < 2; i++ )
  {
     for( int j=0; j < 3; j++ )
     {
        for( int k=0; k < 2; k++ )
        {
           cout << "arr[" << i << "][" << j << "][" << k << "] = ";
           cout << *(*(*(arr+i)+j)+k) << endl;
        }
     }
  }
}
```

Run

```
0xffb8
0xffb8
0xffb8
2
0xffc4
0xffbc
0xffba
3
arr[0][0][0] = 2
arr[0][0][1] = 1
arr[0][1][0] = 3
arr[0][1][1] = 6
arr[0][2][0] = 5
arr[0][2][1] = 3
arr[1][0][0] = 0
arr[1][0][1] = 9
arr[1][1][0] = 2
arr[1][1][1] = 3
arr[1][2][0] = 5
arr[1][2][1] = 8
```

The array `arr` will be stored in memory as shown in Figure 9.11. In the above program, the array name `arr` is the base address of the three dimensional array. The expression `*arr` is the base address of the 0^{th} two dimensional array, `**arr` is the 0^{th} row in the 0^{th} two dimensional array and `***arr` contains the value stored in the 0^{th} column and 0^{th} row of the 0^{th} two dimensional array. The expression `arr+1` is the base address of the 1^{st} two dimensional array, `*arr+1` is the address of the 1^{st} row in the 0^{th} two dimensional array, `**arr+1` gives the address of 0^{th} row and 1^{st} column of a zero dimensional array, `***arr+1` adds 1 to its current value (2) obtained from the 0^{th} element in the 0^{th} row of the 0^{th} two dimensional array. The expression within the `for` loop prints the contents of the three dimensional array in the order in which they are stored in memory.



(a) Three dimensional array (b) Memory organisation for 3-D array

Figure 9.11: Pointer to 3-dimensional arrays

9.11 Pointer Constants

As mentioned earlier, the name of an array holds the starting address of the array. Hence if `arr[3]` is an array of any data type, then the name of the array `arr` is the address of (and does not point to) the 0^{th} element of the array and `arr+1` is the address of the 1^{st} element of the array. If `arr` is a pointer, then `arr+i` cannot be replaced by an expression `arr++` executing `i` times. Using the increment operator with it (the name of the array) is incorrect as the starting address of the array has been placed in the code directly by the compiler, thus making the array name a constant. The array name does not have any storage location allocated unlike a pointer variable which itself has a storage location. Hence, performing an increment operation on the address of the array (which is a constant) is like performing the increment operation; `5++`, which is meaningless. The program `ptrinc.cpp` illustrates these concepts.

```
// ptrinc.cpp: pointers can be incremented but not an array
#include <iostream.h>
void main()
{
   int ia[3] = { 2, 5, 9 };
   int *ptr=ia;
   for( int i = 0; i < 3; i++ )
   {
      //  cout << *(ia++);   error, array address of ia cannot be changed
      cout << " " << *ptr++;  // note: pointer update
   }
}
```

Run

```
 2 5 9
```

In the above program, the elements of the array are accessed using the pointer `ptr` which is assigned the starting address of the array `ia`. The pointer variable `ptr` is incremented every time to point to the next element. The expression `ia++` is incorrect.

9.12 Pointers and String Functions

Like arrays, pointers holding address of strings are widely used for manipulating strings. C++'s library

or user defined functions can be used for manipulating strings. These functions assume the character \0 as the end-of-string indicator and hence, it is not considered as part of a string data. Therefore to store a string of length L, allocate (L+1) bytes of memory. A pointer to the string is passed to these functions instead of the entire string. The program strfunc.cpp illustrates string manipulations using strandard and user defined functions.

```
// strfunc.cpp: user defined string processing functions
#include <iostream.h>
#include <string.h>
// user defined string processing functions prototype
int my_strlen( char *str );
void my_strcpy( char *s2, char *s1 );
void my_strcat( char *s2, char *s1 );
int my_strcmp( char *s1, char *s2 );
void main()
{
   char temp[100], *s1, *s2, *s3;
   cout << "Enter string1: ";
   cin >> temp;
   s1 = new char[ strlen(temp)+1 ];
   my_strcpy( s1, temp );
   cout << "Enter string2: ";
   cin >> temp;
   s2 = new char[ strlen(temp)+1 ];
   my_strcpy( s2, temp );
   cout << "Length of string1: " << my_strlen( s1 ) << endl;
   s3 = new char[ strlen(s1) + my_strlen(s2) + 1 ];
   my_strcpy( s3, s1 );
   my_strcat( s3, s2 );
   cout << "Strings' on concatenation: " << s3 << endl;
   cout << "String comparison using ..." << endl;
   cout << "   Library function: " << strcmp( s1, s2 ) << endl;
   cout << "   User's function: " << my_strcmp( s1, s2 ) << endl;
   delete s1;
   delete s2;
   delete s3;
}
int my_strlen( char *str )
{
   char *ptr = str;
   while( *ptr != '\0' )   // move ptr to end of string
      ++ptr;
   return ptr-str;  // address of last character - starting address = length
}
void my_strcpy( char *s2, char *s1 )
{
   while( *s1 != '\0' )
      *s2++ = *s1++;
   *s2 = '\0'; // copy end of string
}
```

```
void my_strcat( char *s2, char *s1 )
{
   // move end of string
   while( *s2 != '\0' )
      s2++;
   // append s1 to s2
   while( *s1 != '\0' )
      *s2++ = *s1++;
   *s2 = '\0'; // copy end of string
}
int my_strcmp( char *s1, char *s2 )
{
   // compare as long as they are equal
   while( *s1 == *s2 && (*s1 != NULL || *s2 != NULL) )
   {
      s1++;
      s2++;
   }
   return *s1 - *s2;
}
```

Run

```
Enter string1: Object
Enter string2: Oriented
Length of string1: 6
Strings' on concatenation: ObjectOriented
String comparison using ...
   Library function: -16
   User's function: -16
```

9.13 Environment Specific Issues

Pointer variables, like other variables are also allocated memory whenever they are defined. The size of the memory allocated (in bytes) to a pointer variable depends on whether the pointer just holds the offset part of the address, or both the segment and offset values. The memory model in which the program is compiled also influences the size of the pointer variables used in that program. C++ compilers (such as Borland or Microsoft C++) running under DOS environment support six different memory models, each of which determines the amount of memory allocated to the program's data and code (see Table 9.3).

Normally, all pointers defined in a program in the small model contain only the offset part of the address. Such pointers are known as *near pointers,* for which two bytes of memory are allocated. The use of near pointers limits the programmer to access only those memory locations, which lie within a single segment only. (The maximum size of a segment is 64 KB). This limitation can be overcome by the use of pointers, which are capable of holding both the segment as well as the offset part of an address. Such pointers are called *far pointers,* for which four bytes of memory is allocated. It is possible to access any memory location, using far pointers. The far pointers can be defined (even in a small memory model) by using the keyword `far` as follows:

```
int far *ifarptr;  // defines a far pointer to int
```

```
char far *cfarptr; // defines a far pointer to char
```

In the compact and large models, the data area can be more than 64K but any single data structure (like array or structure) should be smaller than 64 KB. For example, if an array is defined as `int far *ary;`, then `ary` will have both a segment and an offset part, but when pointer arithmetic is done, only the offset part is used and not the segment part. If `ary = 0x5437:0xfffe` and it is incremented then `ary` will become `0x5437:0x0000` i.e., the offset part wraps around and the segment part remains unchanged, hence any single data structure should be less than 64 K. However, such limitations are overcome in other memory models such as *huge*.

Memory model	Segment			Pointer	
	Code	Data	Stack	Code	Data
Tiny	64K			near	near
Small	64K	64K		near	near
Medium	1MB	64K		far	near
Compact	64K	1MB		near	far
Large	1MB	1MB		far	far
Huge	1MB	64K each	64K each	far	far

Table 9.3: Memory models

C++ compilers in MS-DOS normally provide three specialized, predefined macros viz., `MK_FP`, `FP_SEG`, and `FP_OFF` for use with `far` and `huge` pointers. The `MK_FP macro` takes two unsigned integer input arguments which are the segment and the offset addresses of the location to be accessed and returns a value that can be used to initialize a `far` or `huge` pointer variable. Here is an example for initializing a `far` pointer variable.

```
char far *cptr;  // define a far pointer variable
. . .
cptr = (char far *) MK_FP( 0xb800, 0x0000 );
```

It causes the `far` pointer `cptr` to point to a byte which resides in segment `0xb800 (in hex)` and at an offset `0x0000 (in hex)`. Note that, the `macro` function `MK_FP` returns a `far` pointer to `void` which must be typecasted suitably before its use.

The macros `FP_SEG` and `FP_OFF` require a `far` pointer as their only input argument, and they return the segment and offset parts of the address contained in that `far` pointer. The three macros mentioned above become available by including the header file `dos.h`.

The program `farptr.cpp` defines a `far` pointer to a character, initializes it with an arbitrary address (say `segment = 0xb800` and `offset= 0x0000`), extracts and prints the segment and offset of the same pointer. It also prints the ASCII character residing at the address `b800:0000`.

```
// farptr.cpp: far pointers and related macros to access display memory
#include <dos.h>
#include <iostream.h>
void main()
```

```
   char ch;
   char far *cptr;   // define far pointer to character
   unsigned int seg_val, off_val;
   // initialize far pointer
   cptr = (char far *) MK_FP( 0xb800, 0x0000 );
   // fetch segment address from far pointer
   seg_val = FP_SEG(cptr);
   // fetch offset address from far pointer
   off_val = FP_OFF(cptr);
   ch = *cptr;
   cout << "Character at 0xb800:0x0000 = " << ch << endl;
   cout << "Segment part of cptr = " << hex << seg_val << endl;
   cout << "Offset part of cptr = " << hex << off_val << endl;
}
```

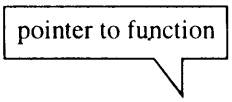
Run

```
Character at 0xb800:0x0000 = S
Segment part of cptr = b800
Offset part of cptr = 0
```

Note: The ASCII character printed by the above program will be the same as the first character on the top left corner of the monitor. It is because the address `b8000:0000` is a location in the video memory, which holds the ASCII value of the character appearing in the top left corner in the text mode.

9.14 Pointers to Functions

A pointer-to-function can be defined to hold the starting address of a function, and the same can be used to invoke a function. It is also possible to pass addresses of different functions at different times thus making the function more flexible and abstract. The syntax of defining a pointer to a function is shown in Figure 9.12.




```
ReturnType (*PtrToFn)(arguments_if_any);
```

Figure 9.12: Syntax of defining pointer to function

The definition of a pointer to a function requires the function's return type and the function's argument list to be specified along with the pointer variable. It should be remembered that the function prototype or definition should be known before its address is assigned to a pointer.

Once a pointer to a function is defined, it can be used to point to any function which matches with the return type and the argument-list stated in the definition of the pointer to a function. Consider a statement such as

```
int (*any_func)(int, int)
```

It defines the variable `any_func` as a pointer to a function. The variable `any_func` can point to any function that takes two integer arguments and returns a single integer value. For instance, it can point to the following functions:

```
int min( int a, int b );
```

```
int max( int a, int b );
int add( int x, int y );
```

Address of a Function

The address of a function can be obtained by just specifying the name of the function without the trailing parentheses. The following statements assign address of the functions to pointer to the function variable `any_func` since prototype of all of them is same:

```
any_func = min;
any_func = max;
any_func = add;
```

Invoking a Function using Pointers

The syntax for invoking a function using a pointer to a function is as follows:

*(*PtrToFn)(arguments_if_any);*

or

PtrToFn(arguments_if_any);

Consider the following pointer to functions

```
int (*pfunc1)( int );
float (*pfunc2)( float, float );
```

If these hold addresses of an appropriate function, the statements

```
(*pfunc1)( 2 );
(*pfunc2)( 2.5, a );
pfunc1( i );
```

invoke functions pointed to by them. The parameters can be constants or variables.

In the definition of pointers to functions, the pointer variable along with the symbol * plays the role of the function name. Hence, while invoking functions using pointers, the function name is replaced by the pointer variable. The program `rfact.cpp` illustrates this concept.

```
// rfact.cpp: pointer to function and its use
#include <iostream.h>
long fact( int num )
{
   if( num == 0 )
      return 1;
   else
      return num * fact( num - 1 );
}
void main( void )
{
   int n;
   long (*ptrfact)(int);      // definition of pointer to function
   ptrfact = fact;            // address of function to pointer assignment
   cout << "Enter the number whose factorial is to be found: ";
   cin >> n;
   long f1 = (*ptrfact)(n);
   cout << "The factorial of " << n << " is " << f1 << endl;
   cout << "The factorial of " << n+1 << " is " << ptrfact(n+1) << endl;
```

Run

```
Enter the number whose factorial is to be found: 5
The factorial of 5 is 120
The factorial of 6 is 720
```

In the above program, a pointer `ptrfact` is defined to point to a function which takes an integer argument and returns an integer value. Then the address of the function `fact` is assigned to the pointer `ptrfact`. The function `fact` computes the factorial of a given positive integer. The function `fact` is invoked using the pointer variable `ptrfact`.

Recursive call to `main()`

When an attempt is made to invoke `main()` within a program, generally compilers generate an error message such as:

```
cannot call main from within the program
```

Because in C++, `main()` cannot be invoked recursively; however it is compiler dependent. The following operations cannot be performed on `main()`:

- `main()` cannot be invoked recursively.
- `main()` cannot be overloaded
- `main()` cannot be declared inline
- `main()` cannot be declared static

The first restriction can be violated by using a pointer to functions. The program `rmain.cpp` invokes `main()` recursively using a pointer to functions.

```
// rmain.cpp: recursive call to main() using a pointer to functions
#include <iostream.h>
void main()
{
   void (*p)();
   cout << "Hello...";
   p = main;
   (*p)();
}
```

Run

```
Hello...Hello...Hello...Hello...Hello...Hello...Hello...Hello...Hello...Hello...
```

The above program generates `Hello...` message indefinite number of times. It stops when stack overflow occurs. In `main()`, the statements

```
p = main;
(*p)();
```

assign the address of `main` to the pointer `p` and transfer control to `main()` using pointer to a function respectively.

Passing Function Address

The address of a function can be passed as an argument to functions, either by a function name or a pointer holding the address of a function. The program `passfn.cpp` illustrates these concepts. It takes two integer parameters and returns the largest and smallest among them.