5.3 Overloaded Functions

An overloaded function appears to perform different activities depending on the kind of data sent to it. It performs one operation on one kind of data but another operation on a different kind.

5.3.1 Different Numbers of Arguments Example:

The starline() function printed a line using 45 asterisks.
The repchar() function used a character and a line length that were both specified when the function was called.
The charline() function that always prints 45 characters but that allows the calling program to specify the character to be printed.
These three functions—starline(), repchar(), and charline()—perform similar activities but have different names.
For programmers using these functions, that means three names to remember and three places to look them up if they are listed alphabetically in an application's.
It would be better to use the same name for all three functions, even though they each have different arguments.
Here's a program, OVERLOAD, that makes this possible:

```
#include <iostream.h>
void repchar(); //declarations
void repchar(char);
void repchar(char, int);
int main()
{ repchar();
  repchar('=');
  repchar('+', 20);
  return 0;
}
void repchar()
{
for(int j=0; j<45; j++)
cout << '*'; // always prints asterisk
cout << endl;
}
void repchar(char ch)
{
for(int j=0; j<45; j++) // always loops 45 times
cout << ch; // prints specified character
cout << endl;
```

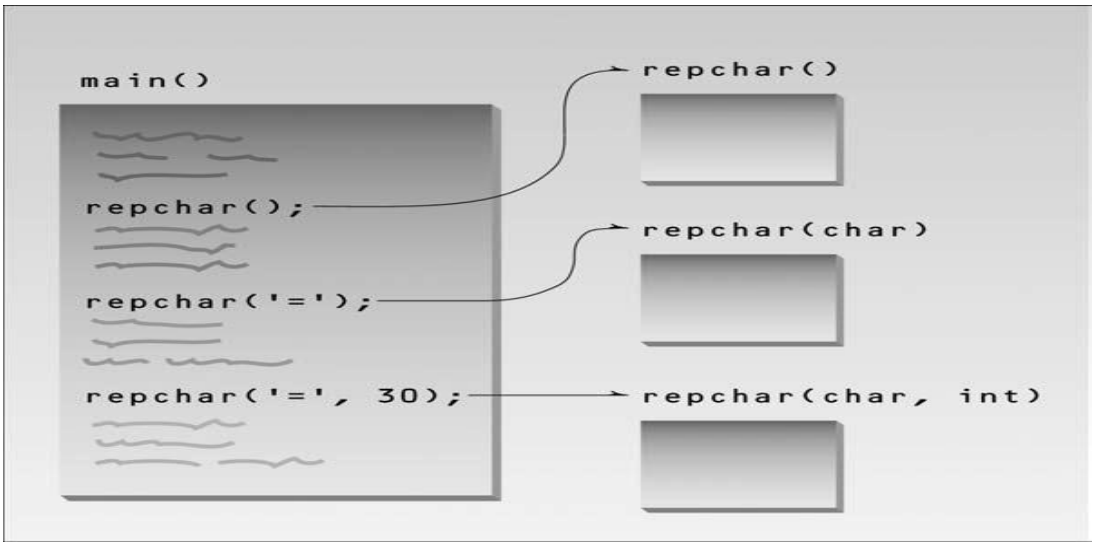
```
}
// repchar()
// displays specified number of copies of specified character
void repchar(char ch, int n)
{for(int j=0; j<n; j++) // loops n times
cout << ch; // prints specified character
cout << endl;
}
```

This program prints out three lines of characters. Here's the output:

```
**********************************************
==============================
++++++++++++++++++++
```

The compiler, seeing several functions with the same name but different numbers of arguments. Which one of these functions will be called depends on the number of arguments supplied in the call.

Figure below shows this process:



5.3.3 Different Kinds of Arguments

In the OVERLOAD example we created several functions with the same name but different numbers of arguments. The compiler can also distinguish between overloaded functions with the same number of arguments, provided their type is different. Here's a program, that uses an overloaded function to display a quantity in feet-and-inches format. The single argument to the function can be either a structure of type Distance or a simple variable of type float. Different functions are used depending on the type of argument.

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

```
{ int feet; float inches; };
void engldisp( Distance );
void engldisp( float );
int main()
{
Distance d1;   float d2;
cout << "\nEnter feet: ";
cin >> d1.feet;
cout << "Enter inches: ";
cin >> d1.inches;
cout << "Enter distance in inches:";
cin >> d2;
cout << "\nd1 = ";
engldisp(d1);
cout << "\nd2 = ";
engldisp(d2);
cout << endl;
return 0;
}
void engldisp( Distance dd ) //parameter dd of type Distance
{  cout << dd.feet << "\'-" << dd.inches << "\"";  }
void engldisp( float dd ) //parameter dd of type float
{
int feet = static_cast<int>(dd / 12);
float inches = dd - feet*12;
cout << feet << "\'-" << inches << "\"";
}
```

Here's some sample interaction with the program:

```
Enter feet: 5
Enter inches: 10.5
Enter entire distance in inches: 76.5
d1 = 5'-10.5"
d2 = 6'-4.5"
```

5.4 Scope and Storage Class

The scope of a variable determines which parts of the program can access it, and its storage class determines how long it stays in existence. Two different kinds of scope are important here: local and file.

- **Variables with local scope are visible only within a block.**
- **Variables with file scope are visible throughout a file.**

A block is the code between an opening brace and a closing brace. Thus a function body is a block.

There are two storage classes: automatic and static.
• Variables with storage class automatic exist during the lifetime of the function in which they're defined.
• Variables with storage class static exist for the lifetime of the program.

5.4.1 Local Variables

All the variables we've used in example programs have been defined inside the function in which they are used: (That is, the definition occurs inside the braces that delimit the function body).

```
void somefunc()
{ int somevar; //variables defined within the function body
  float othervar;
}
```

Variables may be defined inside main() or inside other functions; the effect is the same, since main() is a function. Variables defined within a function body are called local variables
because they have local scope. However, they are also sometimes called automatic variables, because they have the automatic storage class.

5.4.2 Scope

A variable's scope, also called visibility, describes the locations within a program from which it can be accessed. It can be referred to in statements in some parts of the program; but in others, attempts to access it lead to an unknown variable error message. The scope of a variable is that part of the program where the variable is visible. Variables defined within a function are only visible, meaning they can only be accessed, from within the function in which they are defined.
Suppose you have two functions in a program:

```
void somefunc()
{
int somevar; //local variables
float othervar;
somevar = 10; //OK
othervar = 11; //OK
nextvar = 12; //illegal: not visible in somefunc()
}
void otherfunc()
{
```

```
int nextvar; //local variable
somevar = 20; //illegal: not visible in otherfunc()
othervar = 21; //illegal: not visible in otherfunc()
nextvar = 22; //OK
}
```

The variable nextvar is invisible in function somefunc(), and the variables somevar and othervar are invisible in otherfunc().

5.4.3 Global Variables

The next kind of variable is global. While local variables are defined within functions, global variables are defined outside of any function. A global variable is visible to all the functions in a file. More precisely, it is visible to all those functions that follow the variable's definition in the listing. Usually you want global variables to be visible to all functions, so you put their declarations at the beginning of the listing.

Here's a program, in which three functions all access a global variable.

```
// demonstrates global variables
#include <iostream.h>
#include <conio.h> //for getch()
char ch = 'a'; //global variable ch
void getachar();
void putachar();
int main()
{  while( ch != '\r' ) //main() accesses ch
{ getachar();
  putachar();
}
cout << endl;
return 0;
}
void getachar() //getachar() accesses ch
{ ch = getch();}
void putachar() //putachar() accesses ch
{  cout << ch; }
```

5.5 Fundamentals

This program, REPLAY, creates an array of four integers representing the ages of four people. It then asks the user to enter four values, which it places in the array. Finally, it displays all four values.

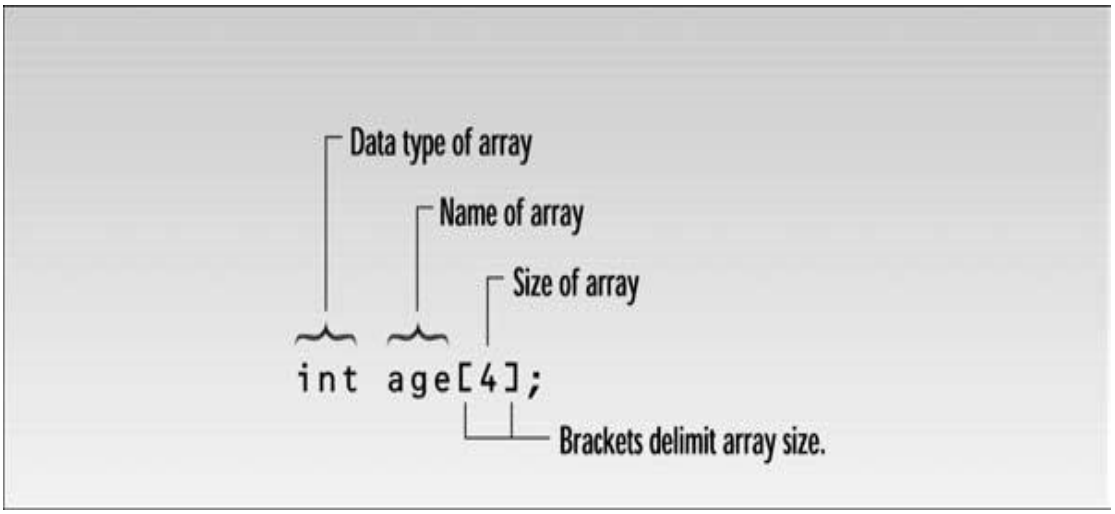
```
// replay.cpp
```

```
// gets four ages from user, displays them
#include <iostream.h>
int main()
{
int age[4];                          //array 'age' of 4 int
for(int j=0; j<4; j++)                //get 4 ages
{cout << "Enter an age: ";
cin >> age[j]; //access array element}
for(j=0; j<4; j++)                   //display 4 ages
cout << "You entered " << age[j] << endl;
return 0;}}
```

Here's a sample interaction with the program:

```
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68
You entered 44
You entered 16
You entered 23
You entered 68
```

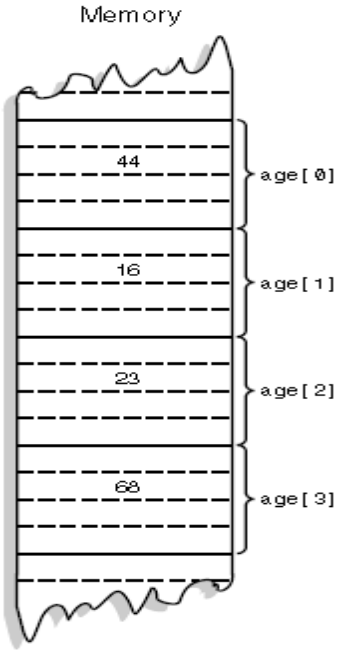
5.5.1 Defining Arrays



Syntax of array definition

5.5.2 Array Elements

The items in an array are called elements (in contrast to the items in a structure, which are called members). As we noted, all the elements in an array are of the same type; only the values vary. Figure bellow shows the elements of the array age.



Array elements.

5.5.3 Accessing Array Elements

In the REPLAY example we access each array element twice. The first time, we insert a value into the array, with the line
cin >> age[j];
The second time, we read it out with the line
cout << "\nYou entered " << age[j];
In both cases the expression for the array element is age[j].This consists of the name of the array, followed by brackets delimiting a variable j. Which of the four array elements is specified by this expression depends on the value of j; age[0] refers to the first element, age[1] to the second, age[2] to the third, and age[3] to the fourth. The variable (or constant) in the brackets is called the array index.

5.5.4 Averaging Array Elements

Here's another example of an array at work. This one, SALES, invites the user to enter a series of six values representing widget sales for each day of the week (excluding Sunday), and then calculates the average of these values. We use an array of type double so that monetary values can be entered.

```
// sales.cpp
// averages a weeks's widget sales (6 days)
#include <iostream.h>
int main()
{
const int SIZE = 6; //size of array
double sales[SIZE]; //array of 6 variables
cout << "Enter widget sales for 6 days\n";
for(int j=0; j<SIZE; j++) //put figures in array
cin >> sales[j];
double total = 0;
for(j=0; j<SIZE; j++) //read figures from array
total += sales[j]; //to find total
double average = total / SIZE; // find average
cout << "Average = " << average << endl;
return 0;}
```

Here's some sample interaction with SALES:

```
Enter widget sales for 6 days
352.64
867.70
781.32
867.35
746.21
189.45
Average = 634.11
```

A new detail in this program is the use of a const variable for the array size and loop limits. This variable is defined at the start of the listing: const int SIZE = 6;