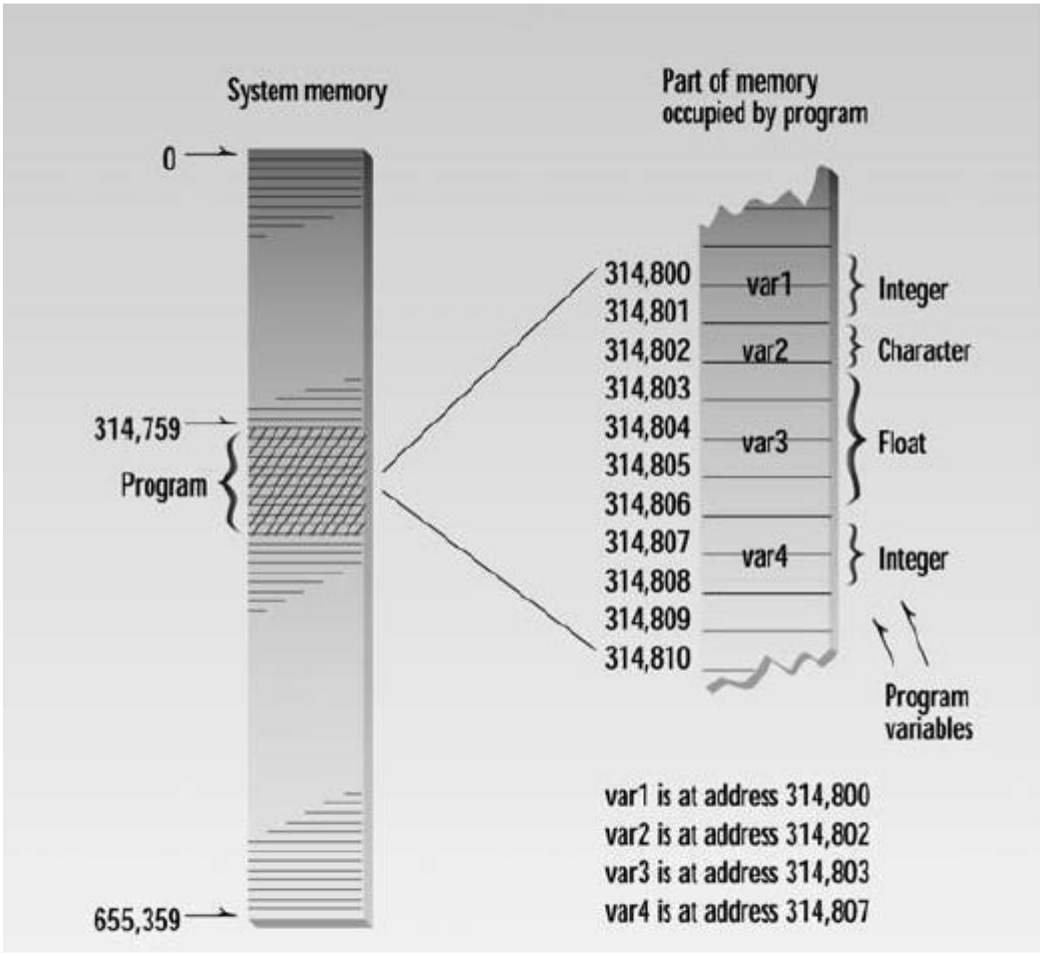
Computer Programming Lab # 03

Pointers and Dynamic Memory Management

Pointers & Addresses:

The ideas behind pointers are not complicated. Here's the first key concept: Every byte in the computer's memory has an address. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on.

Your program, when it is loaded into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address.



What are pointers for? Here are some common uses:

• Accessing array elements
• Passing arguments to a function when the function needs to modify the original argument
• Passing arrays and strings to functions
• Obtaining memory from the system
• Creating data structures such as linked lists

The Address-of Operator &:

You can find the address occupied by a variable by using the address-of operator &. Here’s a short program, VARADDR, that demonstrates how to do this:

```
#include <iostream>
using namespace std;
int main()
{
int var1 = 11; //define and initialize
int var2 = 22; //three variables
int var3 = 33;
cout << &var1 << endl; //print the addresses
cout<< &var2 << endl; //of these variables
cout<< &var3 << endl;
return 0;
}
```

Pointer Variables:

It’s nice to know that we can find out where things are in memory, but printing out address values is not all that useful the potential for increasing our programming power requires an additional idea: variables that hold address values. A variable that holds an address value is called a *pointer variable*, or simply a *pointer*.

What is the data type of pointer variables????

```
#include <iostream>
using namespace std;
int main()
{
int var1 = 11; //two integer variables
int var2 = 22;
cout << &var1 << endl //print addresses
of variables << &var2 << endl << endl;
int* ptr; //pointer to integers i.e. The asterisk means pointer to.
ptr = &var1; //pointer points to var1
cout << ptr << endl; //print pointer
value ptr = &var2; //pointer points to
var2 cout << ptr << endl; //print pointer
value return 0;
}
```

What's wrong with the idea of a general-purpose pointer type that holds pointers to any data type? If we called it type pointer we could write declarations like
Pointer ptr;

Accessing the Variable Pointed To:

```
#include <iostream>
using namespace std;
int main()
{
int var1 = 11; //two integer variables
int var2 = 22;
int* ptr; //pointer to integers
ptr = &var1; //pointer points to var1
cout << *ptr << endl; //print contents of pointer
(11) ptr = &var2; //pointer points to var2
cout << *ptr << endl; //print contents of pointer
(22) return 0;
}
```

When an asterisk is used in front of a variable name, as it is in the *ptr expression, it is called the *dereference operator* (or sometimes the *indirection operator)*. It means *the value of the variable pointed to by*.

You can use a pointer not only to display a variable's value, but also to perform any operation you would perform on the variable directly.

```
#include <iostream>
using namespace std;
int main()
{
int var1, var2; //two integer variables
int* ptr; //pointer to integers
ptr = &var1; //set pointer to address of var1
*ptr = 37; //same as var1=37
var2 = *ptr; //same as var2=var1
cout << var2 << endl; //verify var2 is 37
return 0;
}
```

Remember that the asterisk used as the dereference operator has a different meaning than the asterisk used to declare pointer variables. The dereference operator precedes the variable and means value of the variable pointed to by. The asterisk used in a declaration means pointer to.

```
int* ptr; //declaration: pointer to int
*ptr = 37; //indirection: value of variable pointed to by ptr
```

Pointer to void:

Before we go on to see pointers at work, we should note one peculiarity of pointer data types. Ordinarily, the address that you put in a pointer must be the same type as the pointer. You can't assign the address of a float variable to a pointer to int, for example:

```
float flovar = 98.6;
int* ptrint = &flovar; //ERROR: can't assign float* to int*
```

However, there is an exception to this. There is a sort of general-purpose pointer that can point to any data type. This is called a pointer to void, and is defined like this:

```
void* ptr;        //ptr can point to any data type
```

Such pointers have certain specialized uses, such as passing pointers to functions that operate independently of the data type pointed to.

```
#include <iostream>
using namespace std;
int main()
{
int intvar; //integer variable
float flovar; //float variable
int* ptrint; //define pointer to int
float* ptrflo; //define pointer to float
void* ptrvoid; //define pointer to void
ptrint = &intvar; //ok, int* to int*
// ptrint = &flovar; //error, float* to int*
// ptrflo = &intvar; //error, int* to
float* ptrflo = &flovar; //ok, float* to
float* ptrvoid = &intvar; //ok, int* to
void* ptrvoid = &flovar; //ok, float* to
void* return 0;
}
```

Pointers' Casting:

```
    int ar=1;

    char* b = reinterpret_cast<char*>(&ar);

    cout<<*b;

    int* c= reinterpret_cast<int*>(b);

    cout<<*c;
```

Pointers and Arrays:

Surprisingly, array elements can be accessed using pointer notation as well as array notation.

```
#include <iostream>
using namespace std;
int main()
{ //array
int intarray[5] = { 31, 54, 77, 52, 93 };
for(int j=0; j<5; j++) //for each
element, cout << *(intarray+j) << endl;
//print value return 0;
}
```

The C++ compiler is smart enough to take the size of the data into account when it performs arithmetic on data addresses. It knows that intarray is an array of type int because it was declared that way. So when it sees the expression intarray+3, it interprets it as the address of the fourth integer in intarray, not the fourth byte.

But we want the value of this fourth array element, not the address. To take the value, we use the dereference operator (*). The resulting expression, when j is 3, is *(intarray+3), which is the content of the fourth array element, or 52.

Now we see why a pointer declaration must include the type of the variable pointed to. The compiler needs to know whether a pointer is a pointer to int or a pointer to double so that it can perform the correct arithmetic to access elements of the array. It multiplies the index value by 2 in the case of type int, but by 8 in the case of double.

Pointer Constants and Pointer Variables:

Suppose that, instead of adding j to intarray to step through the array addresses, you wanted to use the increment operator. Could you write *(intarray++)?

The answer is no, and the reason is that you can't increment a constant (or indeed change it in any way). The expression intarray is the address where the system has chosen to place your array, and it will stay at this address until the program terminates. intarray is a pointer constant. You can't say intarray++ any more than you can say 7++.

But while you can't increment an address, you can increment a pointer that holds an address.

```
#include <iostream>
using namespace std;
int main()
{
int intarray[] = { 31, 54, 77, 52, 93 }; //array
int* ptrint; //pointer to int
ptrint = intarray; //points to
intarray for(int j=0; j<5; j++) //for
each element, cout << *(ptrint++) <<
endl; //print value return 0;
}
```

Pointers and Functions:

1. **Passing Simple Variables**

   There are three ways to pass arguments to a function: by value, by reference, and by pointer. If the function is intended to modify variables in the calling program, these variables cannot be passed by value, since the function obtains only a copy of the variable. However, either a reference argument or a pointer can be used in this situation.

   a. Value Pass by Reference

```
#include <iostream>
using namespace std;
int main()
{
void centimize(double&); //prototype
double var = 10.0; //var has value of 10 inches
cout << “var = “ << var << “ inches” << endl;
centimize(var); //change var to centimeters
cout << “var = “ << var << “ centimeters” << endl;
return 0;
}
void centimize(double& v)
{
v *= 2.54; //v is the same as var
}
```

b. Value pass by Pointer

```
#include <iostream>
using namespace std;
int main()
{
void centimize(double*); //prototype
double var = 10.0; //var has value of 10 inches
cout << “var = “ << var << “ inches” << endl;
centimize(&var); //change var to centimeters
cout << “var = “ << var << “ centimeters” << endl;
return 0;
}
void centimize(double* ptrd)
{
*ptrd *= 2.54; //*ptrd is the same as var
}
```

Remember that this is not the variable itself, as it is in passing by reference, but the variable’s address. Because the centimize() function is passed an address, it must use the dereference operator,

```
*ptrd, to access the value stored at this address:
*ptrd *= 2.54; // multiply the contents of ptrd by 2.54
```

A reference is an alias for the original variable, while a pointer is the address of the variable.

2. Passing Arrays

```
#include <iostream>
using namespace std;
const int MAX = 5; //number of array elements
int main()
{
void centimize(double*); //prototype
double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3
}; centimize(varray); //change elements of varray
to cm for(int j=0; j<MAX; j++) //display new array
values cout << “varray[“ << j << “]=”
<< varray[j] << “ centimeters” <<
endl; return 0;
}
```

```
void centimize(double* ptrd)
{
for(int j=0; j<MAX; j++)
*ptrd++ *= 2.54; //ptrd points to elements of varray
}
```

Here's a syntax question: How do we know that the expression *ptrd++ increments the pointer and not the pointer contents? In other words, does the compiler interpret it as *(ptrd++), which is what we want, or as (*ptrd)++? It turns out that * (when used as the dereference operator) and ++ have the same precedence. However, operators of the same precedence are distinguished in a second way: by *associativity*. Associativity is concerned with whether the compiler performs operations starting with an operator on the right or an operator on the left.if a group of operators has right associativity, the compiler performs the operation on the right side of the expression first, then works its way to the left. The unary operators such as * and ++ have right associativity, so the expression is interpreted as *(ptrd++), which increments the pointer, not what it points to. That is, the pointer is incremented first and the dereference operator is applied to the resulting address.

The const Modifier and Pointers:

```
const int* cptrInt; //cptrInt is a pointer to constant int
int* const ptrcInt; //ptrcInt is a constant pointer to int
```

Following the first declaration, you cannot change the value of whatever cptrInt points to, although you can change cptrInt itself. Following the second declaration, you can change what ptrcInt points to, but you cannot change the value of ptrcInt itself. You can remember the difference by reading from right to left, as indicated in the comments. You can use const in both positions to make the pointer and what it points to constant.

Memory Management: new and delete

C++ provides a different approach to obtaining blocks of memory: the new operator. This versatile operator obtains memory from the operating system and returns a pointer to its starting point.

```
char* ptr;
ptr = new char[10];
```

If your program reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system.

delete[] ptr;

```
ptr = new SomeClass; // allocate a single object
. . .
```

```
delete ptr; // no brackets following delete
```

Deleting the memory doesn’t delete the pointer that points to it and doesn’t change the address value in the pointer. However, this address is no longer valid; the memory it points to may be changed to something entirely different. Be careful that you don’t use pointers to memory that has been deleted.

Practice Questions:

1. Suppose you have a main() with three local arrays, all the same size and type (say float). The first two are already initialized to values. Write a function called addarrays() that accepts the addresses of the three arrays as arguments; adds the contents of the first two arrays together, element by element; and places the results in the third array before returning. A fourth argument to this function can carry the size of the arrays. Use pointer notation throughout; the only place you need brackets is in defining the arrays.

2. Write a program that reads a group of numbers from the user and places them in an array of type float. Once the numbers are stored in the array pass this array to function, where the function should average them, find the 3rd maximum and 5th minimum and print the results. Use pointer notations.