Pointers and Strings

CSPROG2
Computer Programming 2 for CS

POINTERS

- **Pointers**

  is the memory address of a variable.

- **Syntax:**

  Type_Name *Pointer_1, *Pointer_2, . . ., *Pojnter_n;

- **Declaring a pointer is just like declaring an ordinary variable of a certain type.**

  int *p1, *p2, v1, v2;

  p1 and p2 are pointers

  v1 and v2 are variable

Table: declaring pointer variable

Declaration	Description
int *p;	pointer that points to a variable of type int
float *p;	pointer that points to a variable of type float
double *p;	pointer that points to a variable of type double
MyType *p;	pointer that points to a variable of type MyType Class

Address of operator

- **& symbol**

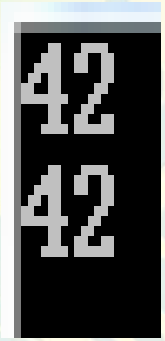
  used to determine the address of a variable.

```
int *p, v;
// set the variable p equal to a pointer
// that points to the variable v
    p = &v;
    *p = 10; // Dereference
```

Source Code example:

```
#include <iostream>
using namespace std;
int main()
{
    int *p, v;
    v = 0;
    p = &v;
    *p = 42;
    cout << v << endl;
    cout << *p << endl;
    system("pause > 0");
    return 0;
}
```

Output:



Dereferencing Operator

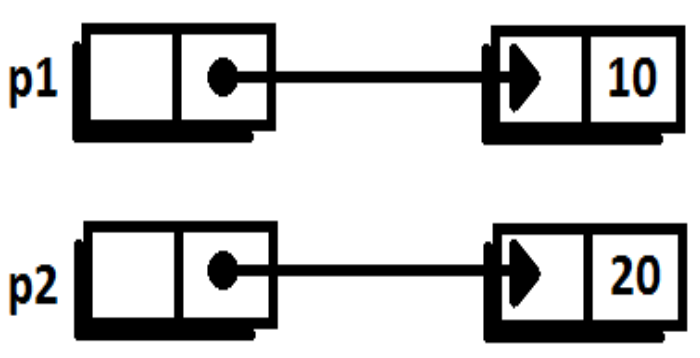
- The * operator in front of a pointer variable produces the variable to which it points. When used this way, the * operator is called the **dereferencing operator**.

- The **&** in front of an ordinary variable produces the address of that variable; that is, it produces a pointer that points to the variable. The **&** operator is simply called the **address-of-operator**.

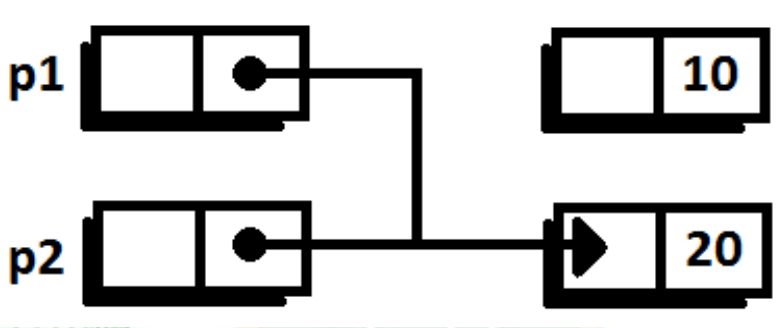
Use of assignment operator with pointer variables

```
p1 = p2;
```

Before:



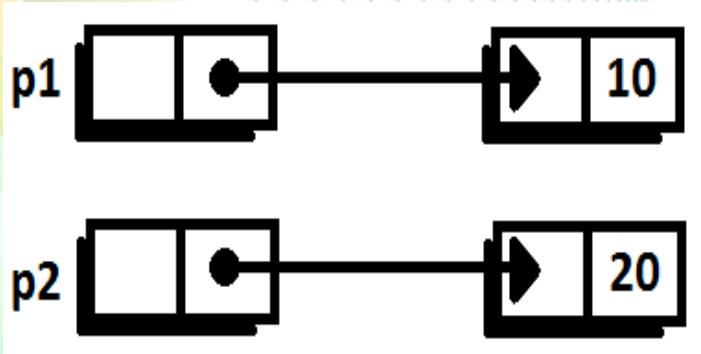
After:



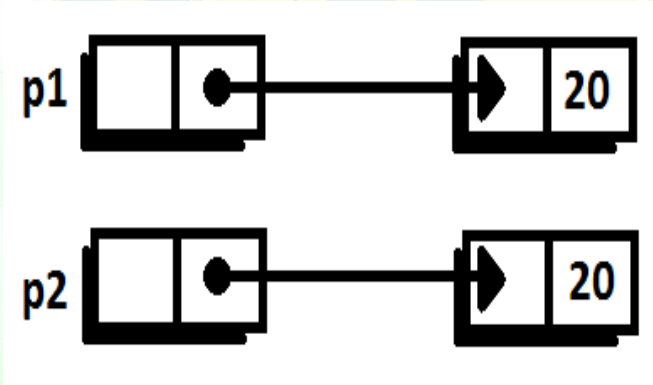
Examples

```
*p1 = *p2;
```

Before:



After:



Source code example:

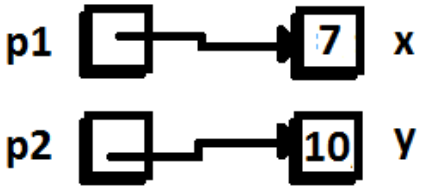
```
#include <iostream>
using namespace std;

int main()
{
    int x(7), y(10), *p1, *p2;
    p1 = &x;
    p2 = &y;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    p1 = p2;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    y = 20;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    x = *p1 + y;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    p2 = &x;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    *p2 = x + y + 5;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    p2 = p1;
    cout << "x:" << x << " y:" << y << " *p1:" << *p1
        << " *p2:" << *p2 << endl;
    system("pause > 0");
    return 0;
}
```

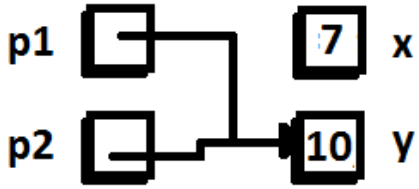
```
x:7 y:10 *p1:7 *p2:10
x:7 y:10 *p1:10 *p2:10
x:7 y:20 *p1:20 *p2:20
x:40 y:20 *p1:20 *p2:20
x:40 y:20 *p1:20 *p2:40
x:65 y:20 *p1:20 *p2:65
x:65 y:20 *p1:20 *p2:20
```

Illustration:

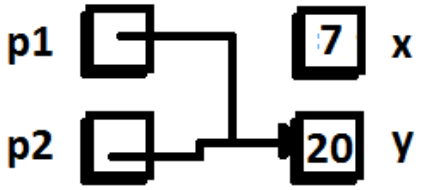
(a) x=7, y=10, p1 = &x, p2 = &y



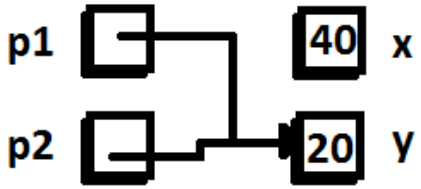
(b) p1 = p2



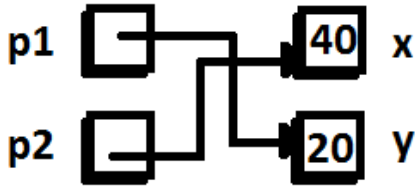
(c) y = 20



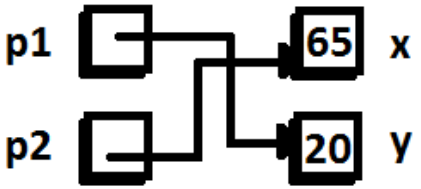
(d) x = *p1 + y



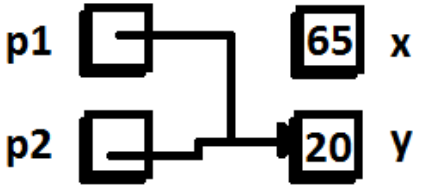
(e) p2 = &x



(f) *p2 = x + y + 5



(g) p2 = p1



The new operator

`new`

Since a pointer can be used to refer to a variable, your program can manipulate variables even if the variables have no identifiers to name them.

The **`new`** operator creates a new dynamic variable of a specified type and returns a pointer that points to this new variable.

Example of `new` operator:

```
p1 = new int;
```

This **new**, nameless variable can be referred to as *p1 (that is, as the variable pointed by p1)

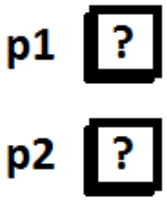
Source code would be:

```
#include <iostream>
using std::cout;
using std::endl;
int main()
{
     int *p1, *p2;
     p1 = new int;
     *p1 = 42;
     p2 = p1;
     cout << "*p1 == " << *p1 << endl;
     cout << "*p2 == " << *p2 << endl;
     *p2 = 53;
     cout << "*p1 == " << *p1 << endl;
     cout << "*p2 == " << *p2 << endl;
     p1 = new int;
     *p1 = 88;
     cout << "*p1 == " << *p1 << endl;
     cout << "*p2 == " << *p2 << endl;
     system ("pause > 0");
     return 0;
}
```

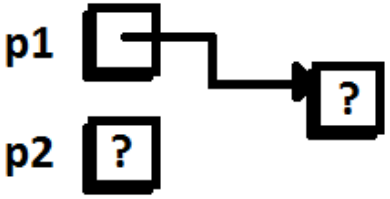
```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
```

Illustration:

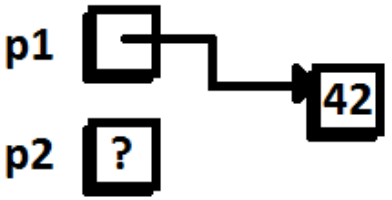
(a) int *p1, *p2;



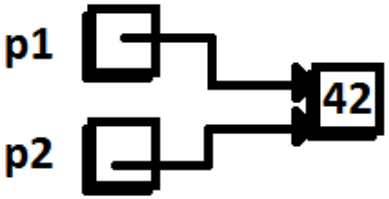
(b) p1 = new int;



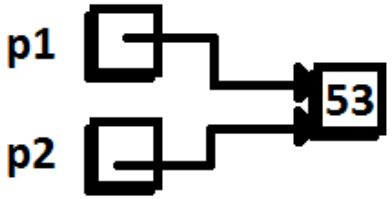
(c) *p1= 42



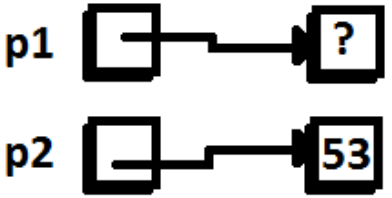
(d) p2 = p1;



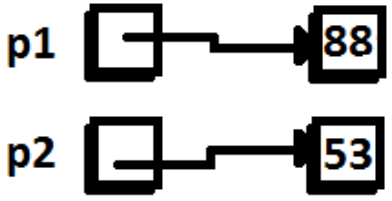
(e) *p2 = 53;



(f) p1 = new int;



(g) *p1 = 88;



The heap

- `heap (freestore)`

It is a special area of memory that is reserved for dynamically allocated variables. Any new dynamic variable created by a program consumes some of the memory in the freestore. If there was insufficient available memory to create the new variable, then new returned a special value named `NULL`.

The delete operator

`delete`

operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore manager so that the memory can be reused. When you apply **`delete`** to a pointer variable, the dynamic variable to which it is pointing is destroyed. At that point, the value of the pointer variable is undefined, which means that you do not know where it is pointing. If pointer variable pointing to the dynamic variable that was destroyed and becomes undefined is called **dangling pointers**.

Type Definitions

- You can assign a name definition and then use the type name to declare variables using typedef keyword.

- **Syntax:**

```
typedef known_type_definition New_Type_Name;
```

Example source code:

- Source code:

```
#include <iostream>
using namespace std;
typedef int* IntPtr;
int main()
{
     IntPtr p1,p2;
     p1 = new int;
     *p1 = 100;
     p2 = p1;
     cout << *p2;
     system("pause>0");
     return 0;
}
```

Ouput:

Pointers as Call-by-Value

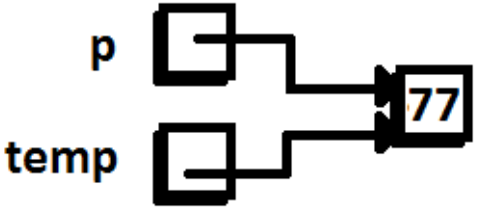
```
#include <iostream>
using namespace std;
typedef int* IntPtr;
void sneaky(IntPtr temp);
int main()
{
    IntPtr p;
    p = new int;
    *p = 77;
    cout << "*p == " << *p << endl;
    sneaky(p);
    cout << "*p == " << *p << endl;
    system("pause > 0");
    return 0;
}
void sneaky(IntPtr temp)
{
    *temp = 99;
    cout << "*temp == " << *temp << endl;
}
```

```
*p == 77
*temp == 99
*p == 99
```

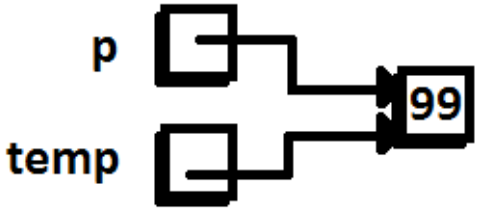
Illustration:

1. Before call to sneaky

2. Value of p is plugged in for temp



3. Change made to *temp



4. After call to sneaky

Dynamic Arrays (Dynamically allocated array)

- Is an array whose size is not specified when you write the program, but is determined while the program is running.

```
int a[10];
int* p;
```

- legal: `p = a;`
- illegal: `a = p;`

Array and pointers

- an array variable is not of type `int*`, but its type is a const version of `int*`. An array variable, like a, is a pointer variable with the modifier `const`, which means that its value cannot be changed.

Sample source code

```
#include <iostream>
using namespace std;
typedef int* IntPtr;
int main()
{
     IntPtr p;
     int a[10];
     int i;

     for(i = 0; i < 10; i++)
          a[i] = i;
     p = a;

     for(i = 0; i < 10; i++)
          cout << p[i] << " ";
     cout << endl;

     for(i = 0; i < 10; i++)
          p[i] = p[i] + 1;

     for(i = 0; i < 10; i++)
          cout << a[i] << " ";
     cout << endl;

     system("pause > 0");
     return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Pointer Arithmetic

d contains the address of the indexed variable `d[0]`. The expression `d + 1` evaluates to the address of `d[1]`, d + 2 is the address of `d[2]`, , and so forth.

```
typedef double* DoublePtr
DoublePtr d;
d = new double[10];
```

Arrays and Pointers

```
for(int i = 0; i < arraysize; i++)
    cout << *(d+1) << " ";
```

is equivalent to:

```
for(int i = 0; i < arraysize; i++)
    cout << d[i] << " ";
```

Example source code:

```
#include <iostream>
using namespace std;
typedef int* IntPtr;
int main()
{
     IntPtr d;
     d = new int[5];
     for(int i=0;i<5;i++)
     {
          d[i] = i * 5;
     }
     for(int i=0;i<5;i++)
     {
          cout << *(d+i) << endl;
     }
     system("pause>0");
     return 0;
}
```

Output: