Comp151

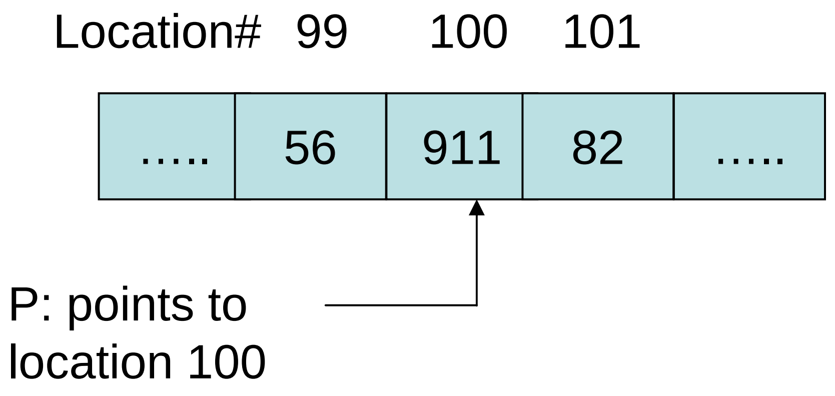
Pointers

Pointer Review +

- Introduction
- `*`and &
- Use of `typedef`
- Dynamic Allocation: `new, delete`
- Dangling pointers
- Memory leakage
- Array = Pointer
- Pointer Arithmetic
- Review: Pointers on Records
- Dynamic Allocation of Arrays

Pointers

- A **pointer** or **pointer variable** is a variable that can reference a memory cell. It does this by storing the location or address of the memory cell.



- Technical questions: If P is a pointer variable,
  - How can we get P to point to a particular memory cell?
  - How can we use the location stored in P to get to the contents of cell to which P points?

Pointer Operations in C++

- Keywords/symbols used are `*`, `&`, `new`, `delete`.

```
int X, Y;    //X and Y are integers
int* P;      // P is an integer pointer variable
```

- The second statement allocates a pointer variable P whose value is undefined but is not NULL. This pointer variable may only point to a memory cell that contains an integer.

```
P = &X;      // Places the address of X into P
             // P points to X.
*P = X;      // Set the value of the memory location
             // pointed to by P to the value stored in X
```

Example

```
   Y = 5;            // variable Y stores value 5
   P = &X;           // P points to memory location of X
   *P = Y;           // same as writing X=Y;
```

At the end of this example

X = 5, Y = 5,

and P points to X.

```
//Program to demonstrate pointers.
//Modified from Savitch Display 11.2
#include <iostream.h>

int main( )
{   int x=10; int y=20;
    int *p1, *p2;

    p1 = &x;
    p2 = &y;
    cout << "x == " << x << endl;
    cout << "y == " << y << endl;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl <<endl;

    *p1 = 50;
    *p2 = 90;
    cout << "x == " << x << endl;
    cout << "y == " << y << endl;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl <<endl;

    p1= p2;
    cout << "x == " << x << endl;
    cout << "y == " << y << endl;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl <<endl;
}
```

x == 10

y == 20

*p1 == 10

*p2 == 20

x == 50

y == 90

*p1 == 50

*p2 == 90

x == 50

y == 90

*p1 == 90

*p2 == 90

Notes

- Read `*P` as **The variable pointed to by `P`**
  Read &X as **The address of X**

- & is the **address of** operator
  `*` is the **dereferencing** operator

- Suppose `P1 = &X` and `P2 = &Y`.
  Then `P1` points to X and P2 points to Y.

        P1 = P2

  does **not** have the same effect as

        *P1 = *P2

  `P1 = P2` means that `P1` now points to Y.
  It does **not** change X.

  `*P1 = *P2` is the same as `X = Y`.

Notes

- The previous example included this:

```
int *p1, *p2;
```

- Not this, which might seem more natural:

```
int* p1, p2;
```

  Why not?

- Consider the difference :

```
(int*) p1, p2;
int(* p1), p2;
```

- The first is more logical since it groups the type information, but the second is how C++ interprets the code.

- Strongly recommended: use this cleaner convention:

```
int* p1; int* p2;
```

- Strongly recommended: define only one variable per line:

```
int* p1;
int* p2;
```

- Or, alternatively, use **`typedef`**…

- There is a classic error to watch out for when using pointers. It is the difference between the following two lines

```
int* P, Q;            // P is a pointer and Q an int
int *P, *Q;           // P and Q are both pointers
```

- One way to avoid this error is to use the `typedef` command which permits you to define new type names, e.g.

```
typedef double distance; // distance is a new name for double distance miles;
```

  is the same as writing

```
double miles;
```

  This means that instead of writing

```
int *P, *Q;
```

- we can write

```
typedef int* IntPtr;          // new name for pointers to ints
IntPtr P, Q;                  // P and Q are both pointers
```

Static and Dynamic Allocation Of Memory

- The fragment

  int X, Y;   *// X and Y are integers*

  int* P;   *// P is an integer pointer variable*

  allocates memory for X, Y and P at compilation time. This is called **static allocation**.

- Memory may also be allocated at execution time. This is known as Dynamic Allocation. For example

  P = **new** int;

  allocates a new memory cell that can contain an integer and points P to it.

```
//Program to demonstrate pointers
//and dynamic variables.
//Modified from Savitch Display 11.2
#include <iostream.h>

int main( )
{
   int* p1; int* p2;

   p1 = new int;
   *p1 = 10;
   p2 = p1;
   cout << "*p1 == " << *p1 << endl;
   cout << "*p2 == " << *p2 << endl <<endl;

   *p2 = 30;
   cout << "*p1 == " << *p1 << endl;
   cout << "*p2 == " << *p2 << endl <<endl;

   p1 = new int;
   *p1 = 40;
   cout << "*p1 == " << *p1 << endl;
   cout << "*p2 == " << *p2 << endl <<endl;
}
```

```
*p1 == 10
*p2 == 10

*p1 == 30
*p2 == 30

*p1 == 40
*p2 == 30
```

- A special area of memory, the heap is reserved for dynamic variables. To create a new dynamic variable the system allocates space from the heap. If all the memory is used up and new is unable to allocate memory then it returns the value `NULL`.

- In a real programming situation you should always check for this error.

```
int* p;
p = new int;

if (p == NULL)
{
    cout << "Memory Allocation“ << endl;
    exit(1);
}
```

- `NULL` is actually the value 0 but we think of it as something special because we will have use for a special “empty” pointer later.

- The value of `NULL` is defined in `stddef.h` which should be included in any program using `NULL`.

- The system has a limited amount of space on the heap. So as not to use it up it is a good idea to return unused dynamic memory to the heap. If `P` is a pointer this can be done using the complimentary command

  delete(P);

  which deletes the memory cell to which `P` points. <u>It does not modify `P`</u>. After executing `delete(P)` the value of `P` is undefined.

```
//Program to demonstrate delete
//Pointer3.cpp
#include <iostream.h>

typedef int* IntPtr;
int main( )
{   int x= 20;
    IntPtr p;
    p = new int;
    *p = 30;
    cout << "*p == " << *p;
    cout << " <---> x == " << x << endl;
    delete p;                                   //delete what p points to, but not p itself!
    p = &x;
    cout << "*p == " << *p;
    cout << " <---> x == " << x << endl;
}
```

Output:

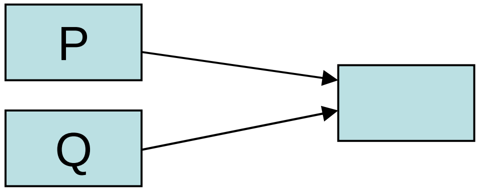
```
*p == 30 <---? x == 20
*p == 20 <---? x == 20
```

The Dangling Pointer

- Be careful that when you use delete P you are not erasing a location that some other pointer Q is pointing to.

```
int* P;    // P is an integer pointer variable
int* Q;    // Q is an integer pointer variable
P = new int;
Q = P;
```

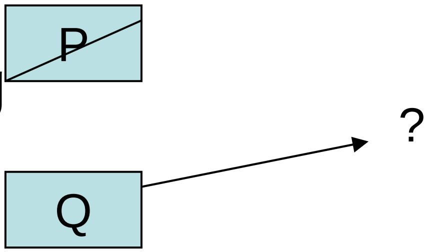
creates



but then executing

```
delete P;
P = NULL;
```

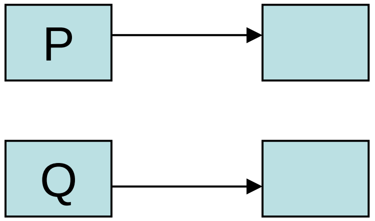
leaves Q dangling



Memory Leakage

- An associated problem is losing all pointers to an allocated memory location. When this happens the memory can never be deallocated and is lost, i.e., never returned to the heap.

```
int* P;        // P is an integer pointer variable
int* Q;        // Q is an integer pointer variable
P = new int;
Q = new int;
```

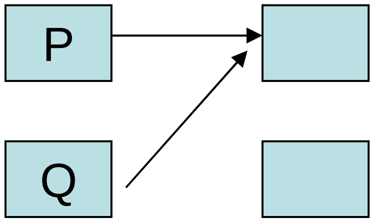


creates

but then executing

```
Q = P;
```

leaves the location previously pointed to by `Q` lost.

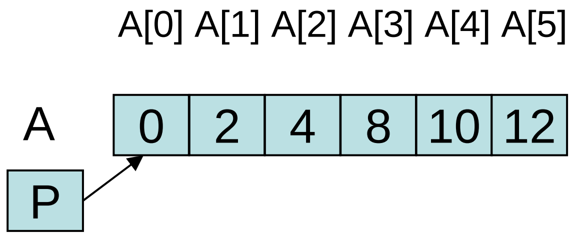


Arrays and Pointers

- An array name is actually a <u>pointer</u> to the beginning of the array !

```
int A[6] ={0,2,4,8,10,12}; //defines an array of inegers
int* P;
P = A;        // P points to A[0]
```

yields

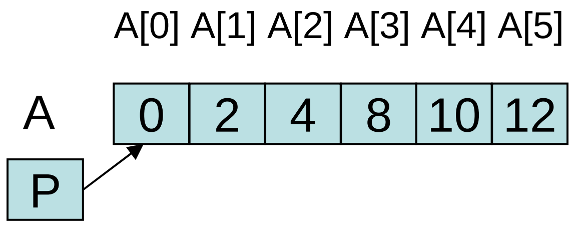


- Since array names and pointers are equivalent we can also use P as the array name. For example

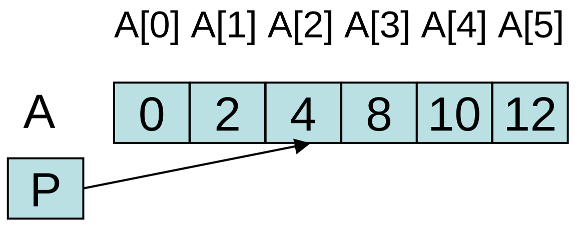
```
P[3] = 7;
```

is equivalent to

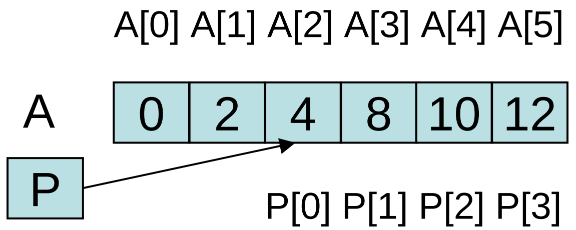
```
A[3] = 7;
```



Starting with the above and executing `P = &(A[2])` yields



but now P[0] refers to A[2], P[1] to A[3], ... , eg.

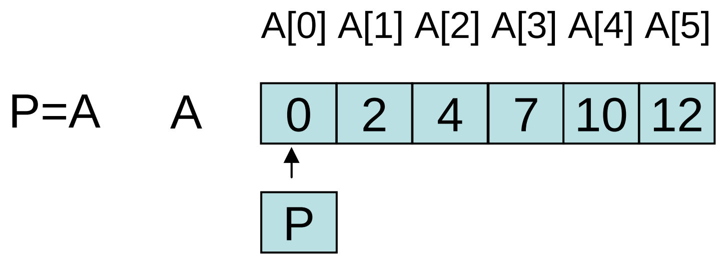


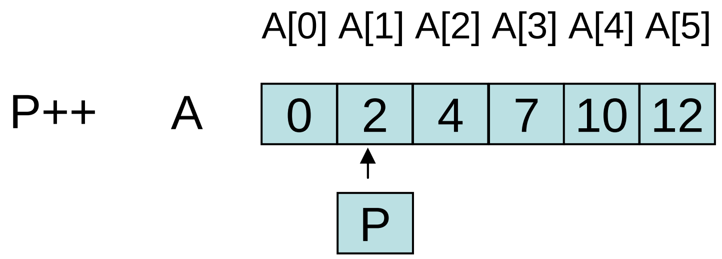
Pointer Arithmetic

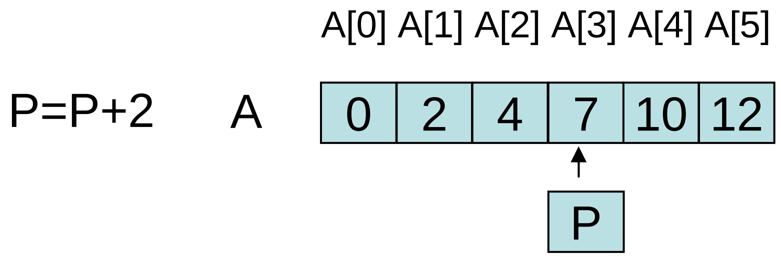
- Arithmetic on pointers has a different meaning than arithmetic on “numbers”. Adding an integer `i` to `P` says that `P` should be advanced `i` data items:

```
SomeType *P;        // Set P to be a pointer to some type
P + i;              // increment P by i*sizeof(Type) bytes
```

Examples:

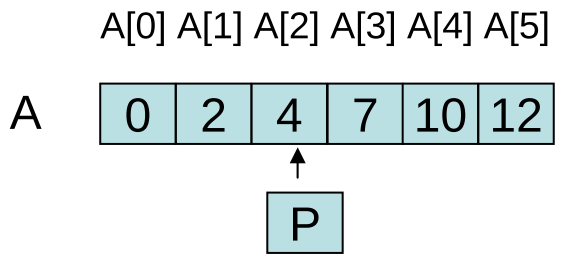




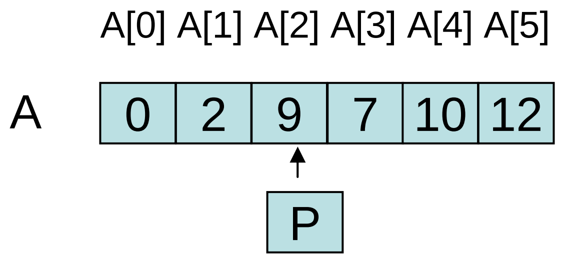


More examples

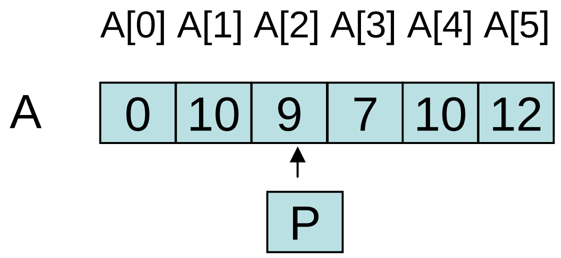
P=A+2



P=(P+1)+2



(P-1)=(P+2)



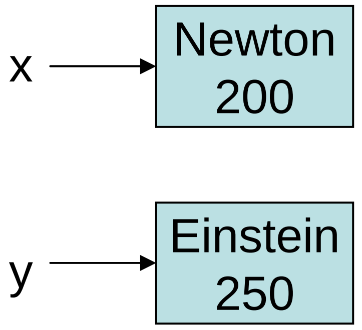
Pointers on Records: Dynamic Memory Allocation

```
#ifndef IQ1_H
#define IQ1_H
#include <iostream.h>

class IQ {
private:
    char name[20];
    int score;
public:
    IQ(const char* s, int k)
    {
        strcpy(name, s);
        score = k;
    }
    void smarter(int k)
    {
        score += k;
    }
    void print() const
    {
        cout << "( " << name << " , " << score << " )" << endl;
    }
};
#endif        // IQ1_H
```

```
#include <iostream.h>
#include "iq1.h"

int main()
{
    IQ* x = new IQ("Newton", 200);
    IQ* y = new IQ("Einstein", 250);
    x->print();
    y->print();
    return 0;
}
```



Pointers on Records: Indirect Addressing

```
#include <iostream.h>
#include "iq1.h"

int main()
{
    int choice;
    IQ x("Newton", 200);
    IQ y("Einstein", 250);
    cin >> choice;
    if (choice == 1)
    {
      x.smarter(50);
      x.print();
    }
    else
    {
      y.smarter(50);
      y.print();
    }
    return 0;
}
```

```
#include <iostream.h>
#include "iq1.h"

int main()
{
    int choice;
    IQ* iq_ptr;
    IQ x("Newton", 200);
    IQ y("Einstein", 250);
    cin >> choice;
    if (choice == 1)
        iq_ptr = &x;
    else
        iq_ptr = &y;
    iq_ptr->smarter(50);
    iq_ptr->print();
return 0;
}
```

Pointers on Records: Indirect Addressing

```
#include <iostream.h>
#include "iq1.h"

int main()
{
    int choice;
    IQ x("Newton", 200);
    IQ y("Einstein", 250);
    cin >> choice;
    if (choice == 1)
    {
      x.smarter(50);
      x.print();
    }
    else
    {
      y.smarter(50);
      y.print();
    }
    return 0;
}
```

```
#include <iostream.h>
#include "iq1.h"

int main()
{
    int choice;
    IQ* iq_ptr;
    IQ x("Newton", 200);
    IQ y("Einstein", 250);
    cin >> choice;
    if (choice == 1)
        iq_ptr = &x;
    else
        iq_ptr = &y;
    (*iq_ptr).smarter(50);
    (*iq_ptr).print();
return 0;
}
```

Dynamic Allocation of Arrays

- **`new T[n]`** will allocate an array of n objects of type T : It will return a pointer to the start of the array.
- **`delete [ ] p`** will destroy the array to which p points and return the memory to the heap. p must point to the front of a dynamically allocated array. If it does not, the resulting computation will be ambiguous, i.e., it will depend upon what compiler you are using and what data you are inputting. You might get a run-time error, a wrong answer . . ..

```
//Program to demonstrate dynamic arrays
#include <iostream.h>
int main( )
{
    // dynamically allocate array
    int* A = new int[6];

    A[0] = 0; A[1] = 1; A[2] = 2;
    A[3] = 3; A[4] = 4; A[5] = 5;
    cout << "A[1] = " << A[1] << endl;

    // delete the array
    delete [] A;
}
```

```
// Program to demonstrate an illegal delete on dynamic arrays

#include <iostream.h>

int main( )
{
    // dynamically allocate array
    int* A = new int[6];

    A[0] = 0; A[1] = 1; A[2] = 2;
    A[3] = 3; A[4] = 4; A[5] = 5;

    int* p = A + 2;

    cout << "A[1] = " << A[1] << endl;

    // this is ILLEGAL!
    delete [] p;

    // the result of this will depend upon the particular compiler
    cout << "A[1] = " << A[1] << endl;
}
```

Dynamic Allocation of Arrays

- The dimension of the dynamically allocated array does not have to be a constant. It can be an expression evaluated at run time.

```
// Program to demonstrate dynamic arrays with run-time evaluation of dimension
#include <iostream.h>
#include <stdlib.h>                    // needed for atoi()

int main( int argc, char* argv[] )
{   // get dimension of array and allocate it
    int dim = atoi(argv[1]);
    int* A = new int[dim];

    // initialize and print array contents
    for (int i=0; i< dim; i++)
                A[i]= i;
    for (int i=0; i< dim; i++)
                cout << "A[" << i << "] = " << A[i] << endl;

    // delete array
    delete [] A;
}
```

```
557> a.out 4
A[0] = 0
A[1] = 1
A[2] = 2
A[3] = 3
558> a.out 7
A[0] = 0
A[1] = 1
A[2] = 2
A[3] = 3
A[4] = 4
A[5] = 5
A[6] = 6
```