H.O.#6
Fall 2015
Gary Chan

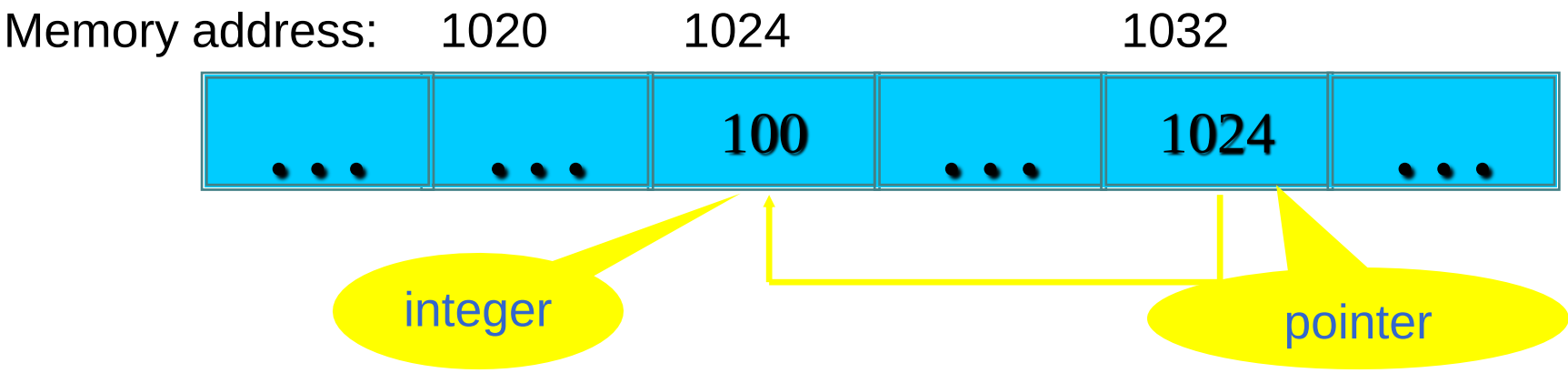
Pointers, Dynamic Objects and struct

Topics

- Pointers
  - Memory addresses
  - Declaration
  - Dereferencing a pointer
  - Pointers to pointer
- Static vs. dynamic objects
  - `new` and `delete`
- `Struct`

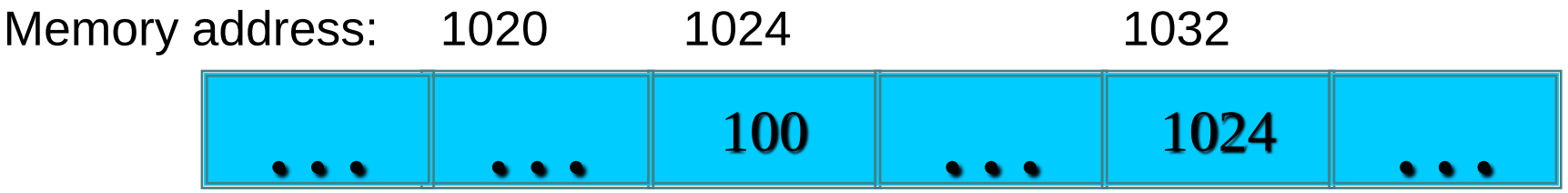
Pointers

- A pointer is a variable used to store the address of a memory cell.
- We can use the pointer to reference this memory cell



Computer Memory

- A variable is in fact a portion of memory to store a determined value
- Each variable is assigned a memory slot (the size depends on the data type) and the variable's data is stored there




```
int a = 100;
```

Variable a's value, i.e., 100, is stored at memory location 1024

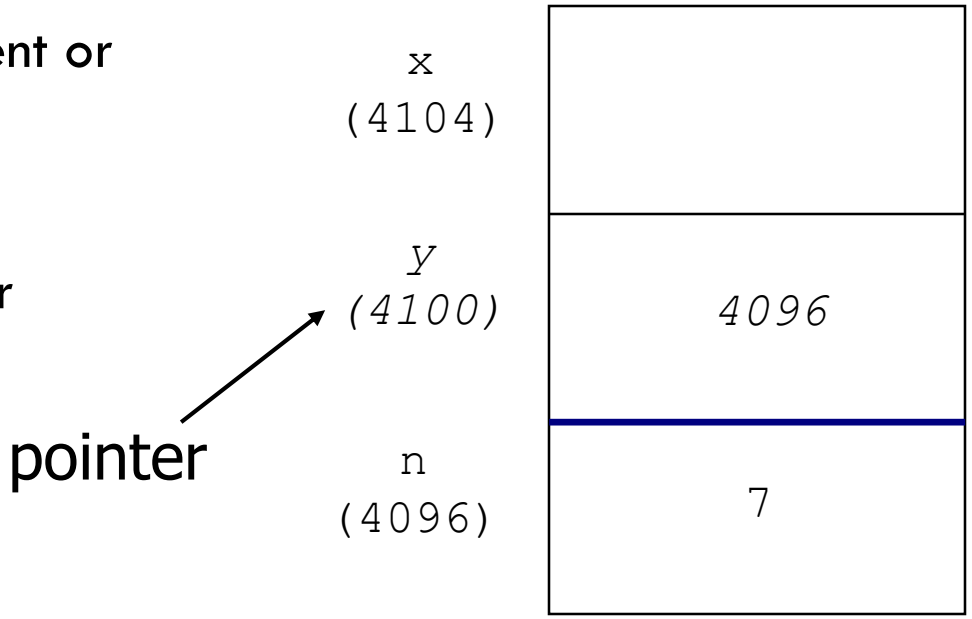
Pointer Types

- Pointer
  - C++ has pointer types for each type of object
    - Pointers to `int` objects
    - Pointers to `char` objects
    - Pointers to user-defined objects

      (e.g., `RationalNumber`)
  - Even pointers to pointers
    - Pointers to pointers to `int` objects

Pointers

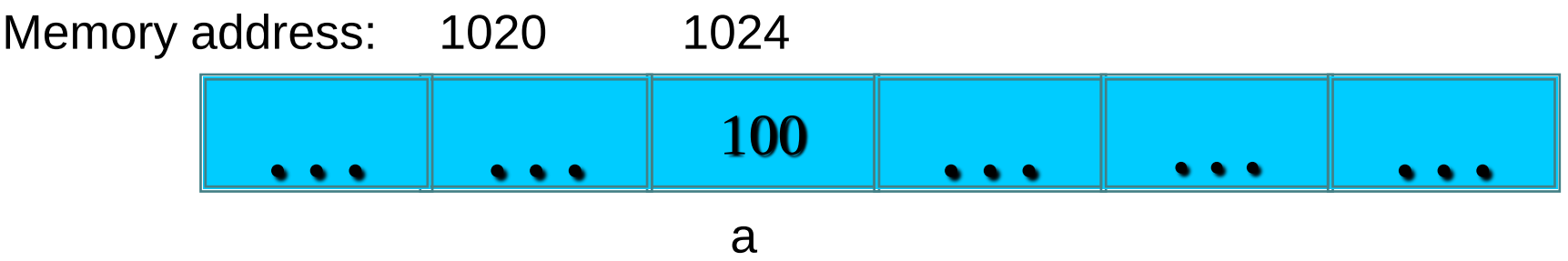
- A *pointer* is a variable which contains addresses of other variables
- A pointer points to an element or an *array* of a certain type
- Accessing the data at the contained address is called "dereferencing a pointer" or "following a pointer"




```
int main(){

  int n=7;
  int * y = &n;

}
```

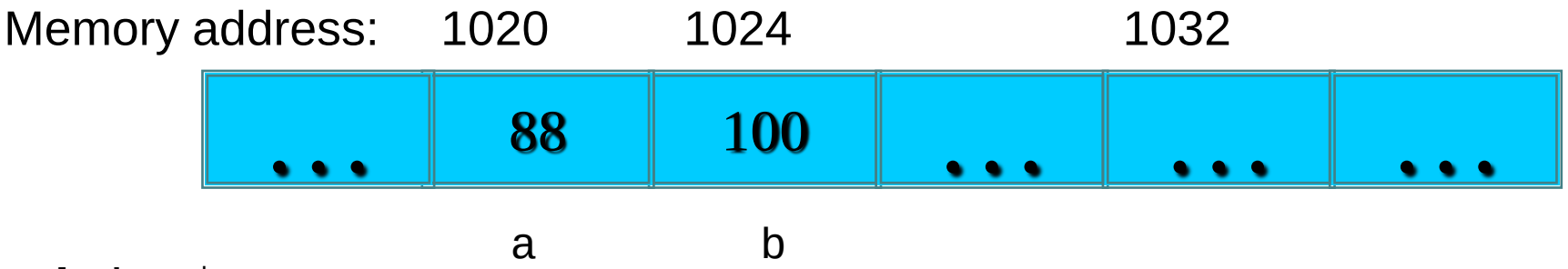
Address Operator &

- *The "address of" operator* (&) gives the memory address of the variable
  - **Usage: `&variable_name`**




```
int a = 100;
//To get the value, use the variable name
cout << a;    //prints 100
//To get the memory address, add the address
//operator before the variable name
cout << &a; //prints 1024
```

Address Operator &




```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

Result is:
The address of a is: 1020
The address of b is: 1024

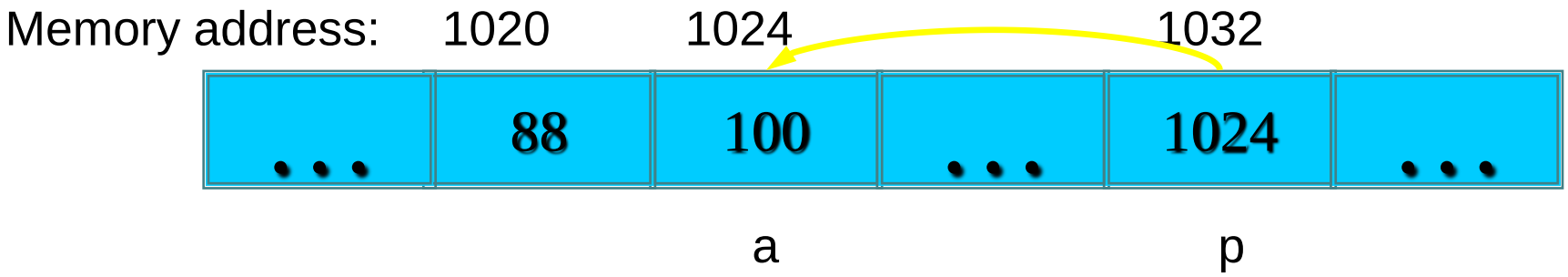
Pointer Variable

- A pointer variable is a specific box for storing a memory address
- Declaration of Pointer variables

```
type* pointer_name;
//or
type *pointer_name;
```

Where *type* is the type of data pointed to (e.g. int, char, double)

Pointer Variables



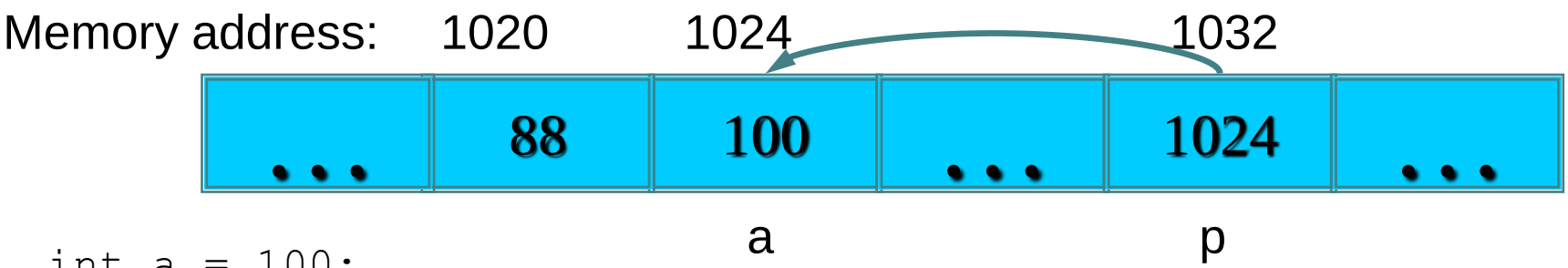

```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

Result is:
100 1024
1024 1032

- The value of pointer `p` is the address of variable `a`
- A pointer is also a variable, so it has its own memory address

Dereference Operator *

- We can access to the value stored in the variable pointed to by preceding the pointer with the "star" dereference operator (*),




```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

Result is:
100
1024
1024 100
1032

Don't get confused

- Declaring a pointer means only that it is a pointer: `int *p;`
- Don't be confused with the dereference operator, which is also written with an asterisk (*). They are simply two different tasks represented with the same sign

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b;     // p2 points to b
p2 = p1;     // p2 points to a
b = *p3;     //assign c to b
*p2 = *p3;   //assign c to a
cout << a << b << c;
```

Result is:
888

Pointer Example

```
#include <iostream>
using namespace std;
int main (){
  int value1 = 5, value2 = 15;
  int *p1, *p2;
  p1 = &value1; // p1 = address of value1
  p2 = &value2; // p2 = address of value2
  *p1 = 10;     // value pointed to by p1=10
  *p2 = *p1;    // value pointed to by p2= value
                // pointed to by p1
  p1 = p2;        // p1 = p2 (pointer value copied)
  *p1 = 20;     // value pointed to by p1 = 20
  cout << "value1==" << value1 << "/ value2==" << value2;
  return 0;
}
```

Result is

value1==10 / value2==20

Another Pointer Example

```
int a = 3;
char s  = 'z';
double d = 1.03;
int *pa = &a;
char *ps = &s;
double *pd = &d;

cout << sizeof(pa) << sizeof(*pa)
     << sizeof(&pa) << endl;
cout << sizeof(ps) << sizeof(*ps)
     << sizeof(&ps) << endl;
cout << sizeof(pd) << sizeof(*pd)
     << sizeof(&pd) << endl;
```

848
818
888

Traditional Pointer Usage

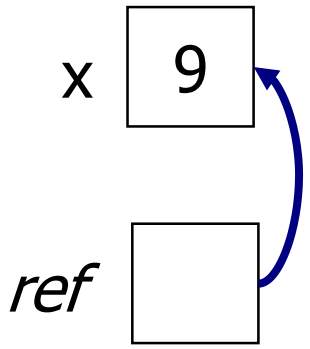
```
void IndirectSwap(char *Ptr1, char *Ptr2){
  char temp = *Ptr1;
  *Ptr1 = *Ptr2;
  *Ptr2 = temp;
}
int main() {
  char a = 'y';
  char b = 'n';
  IndirectSwap(&a, &b);
  cout << a << b << endl;
  return 0;
}
```

Pointer vs. Reference

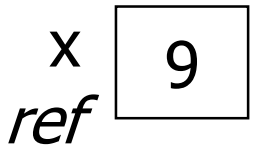
References are an additional name to an existing memory location

If we wanted something called “ref” to refer to a variable x:

Pointer:



Reference:



Pass by Reference: Another Way of Implementation

```
void IndirectSwap(char& y, char& z) {
  char temp = y;
  y = z;
  z = temp;
}
int main() {
  char a = 'y';
  char b = 'n';
  IndirectSwap(a, b);
  cout << a << b << endl;
  return 0;
}
```

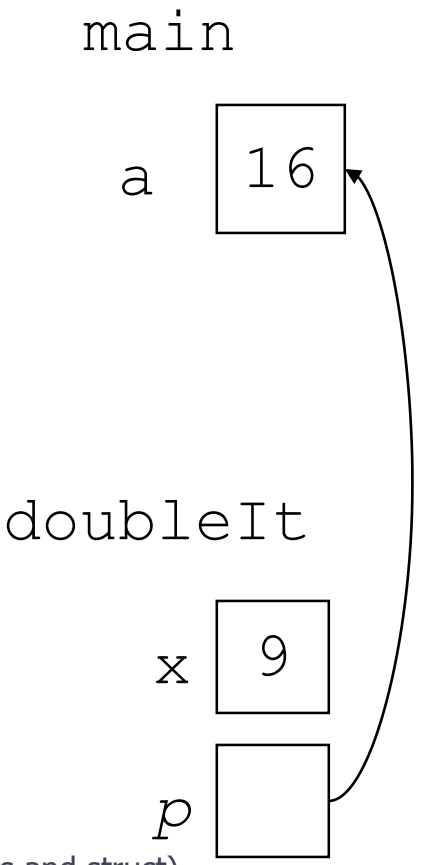
A Pointer Example

The code

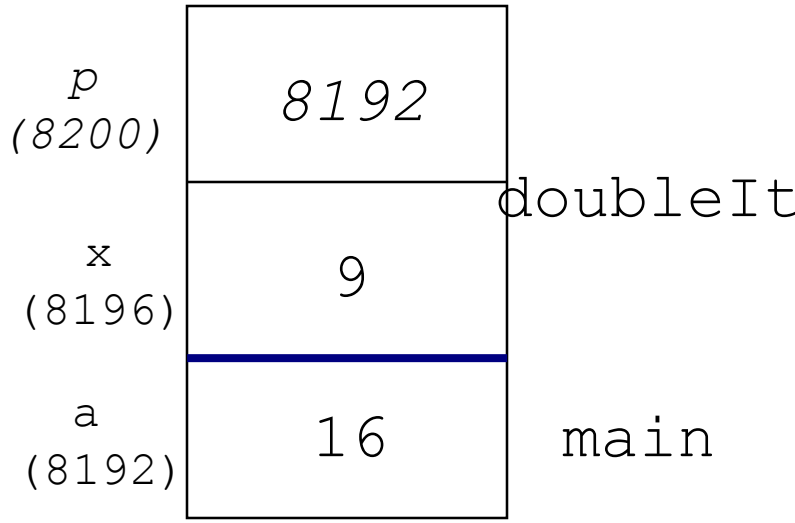
```
void doubleIt(int x,
              int * p)
{
   *p = 2 * x;
}
int main(int argc, const
   char * argv[])
{
   int a = 16;
   doubleIt(9, &a);
   return 0;
}
```

a gets 18

Box diagram



Memory Layout



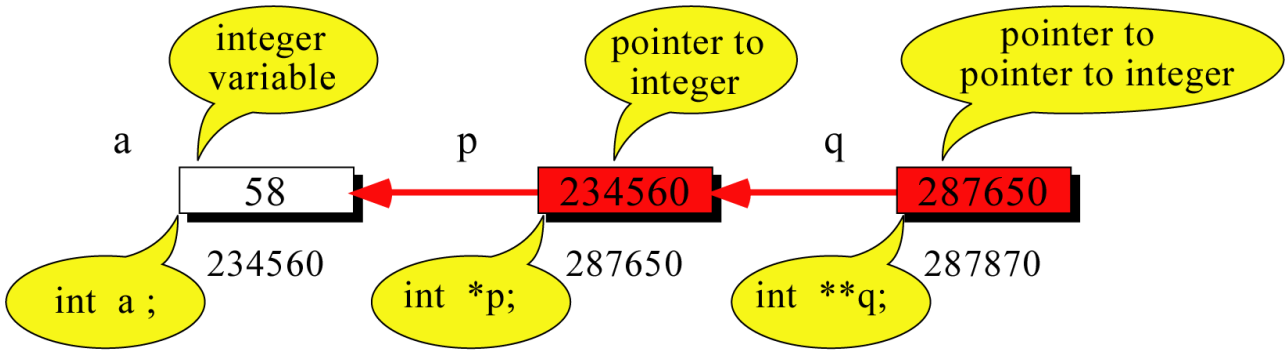
Pointer vs. Reference

- A pointer needs NOT be initialized while defining, but a reference variable should always refer to some other object.
- A pointer can be assigned a new value to point at a different object, but a reference variable always refers to the *same* object. Assigning a reference variable with a new value actually changes the value of the referred object.

```
int * p;  // uninitialized pointer, ok
int m = 10;
int & j = m; //valid, but NOT int &j;
p = &m; //p now points at m
int n = 12;
j = n; // the value of m is set to 12. But j still refers to m, not to n.
cout << "value of m = " << m <<endl; //value of m printed is 12

n = 36;
cout << "value of j = " << j << endl; //value of j printed is 12

p = &n;
```

Pointer to Pointer

```
// Local Declarations
int         a ;
int         *p ;
int         **q ;
```

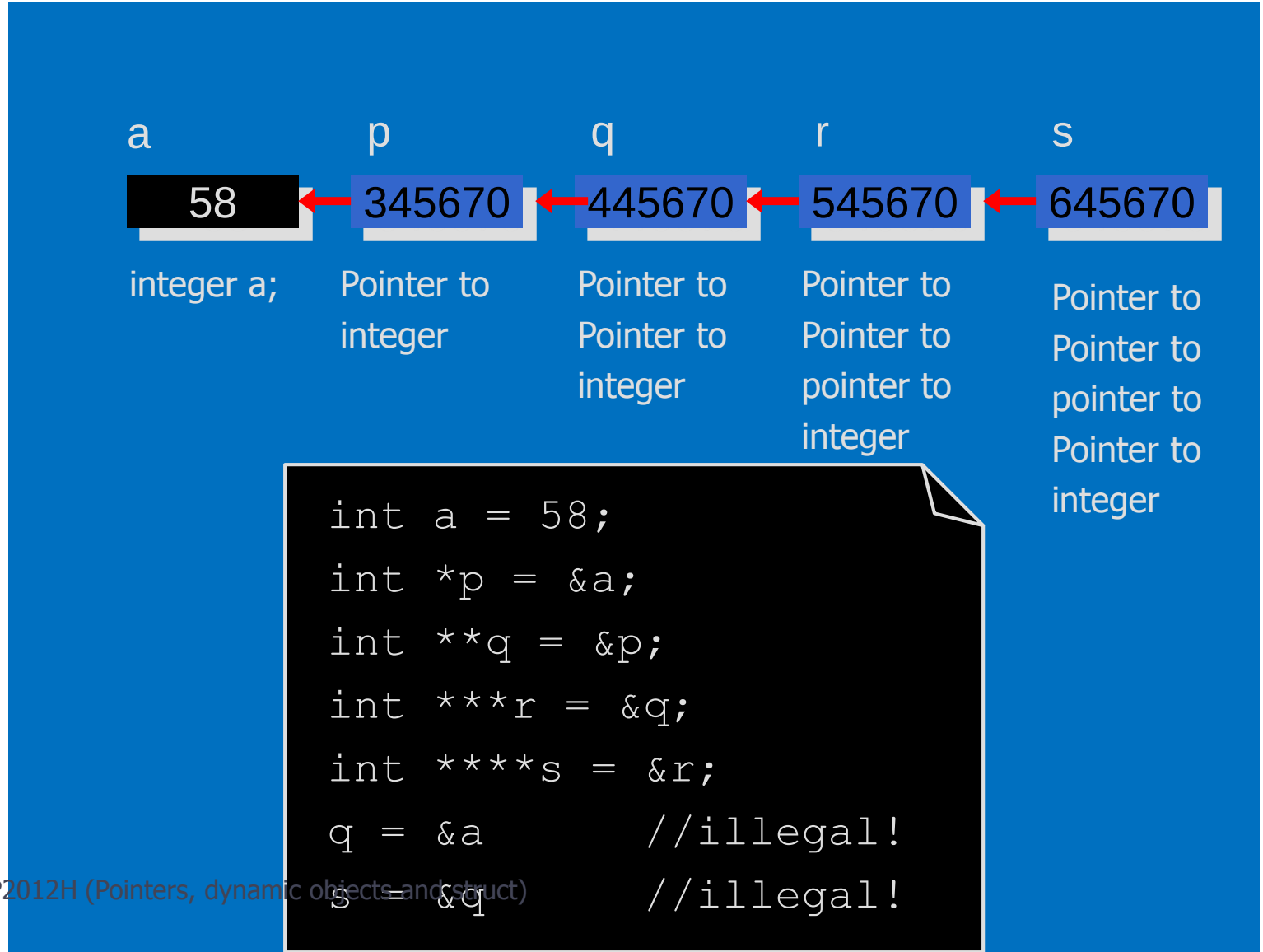


What is the output?

58 58 58

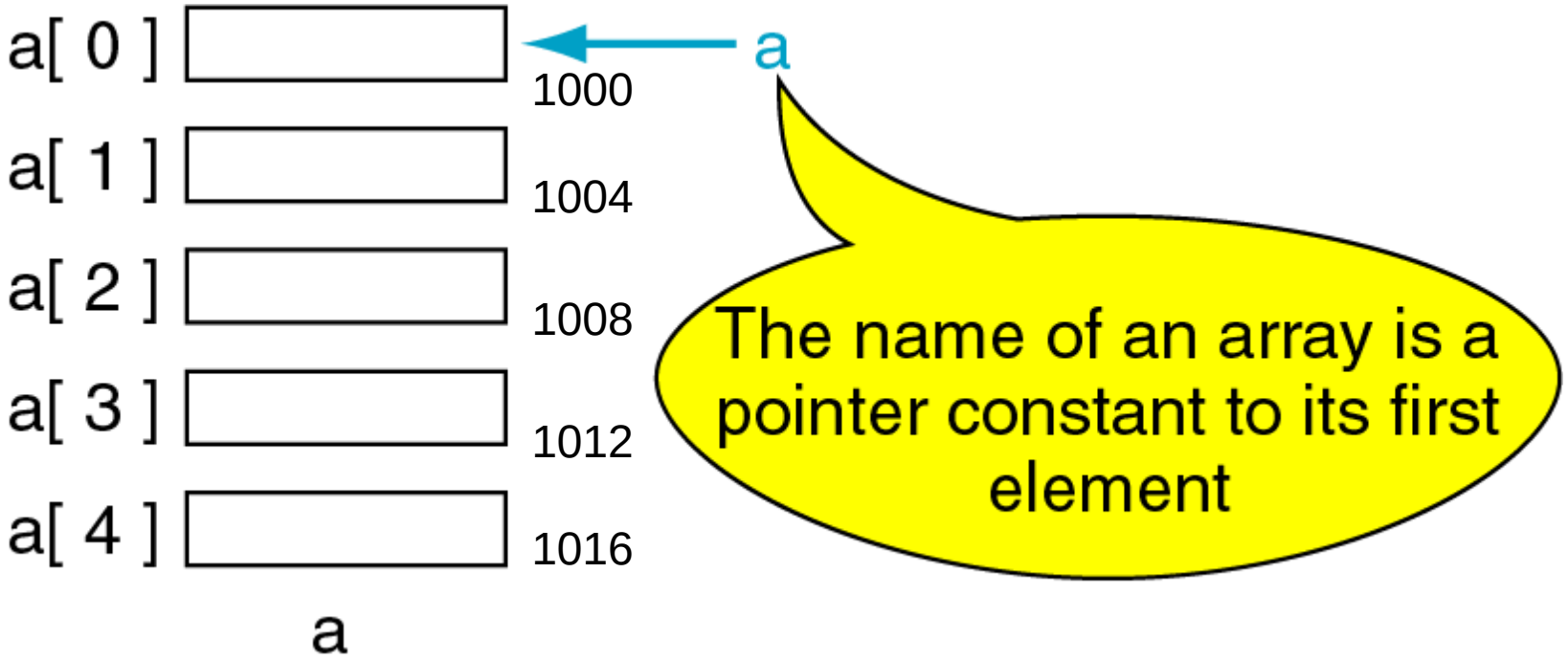
```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<        a << " ";
cout <<       *p << " ";
cout <<      **q << " ";
```

More Pointer to Pointer



Pointers and Arrays

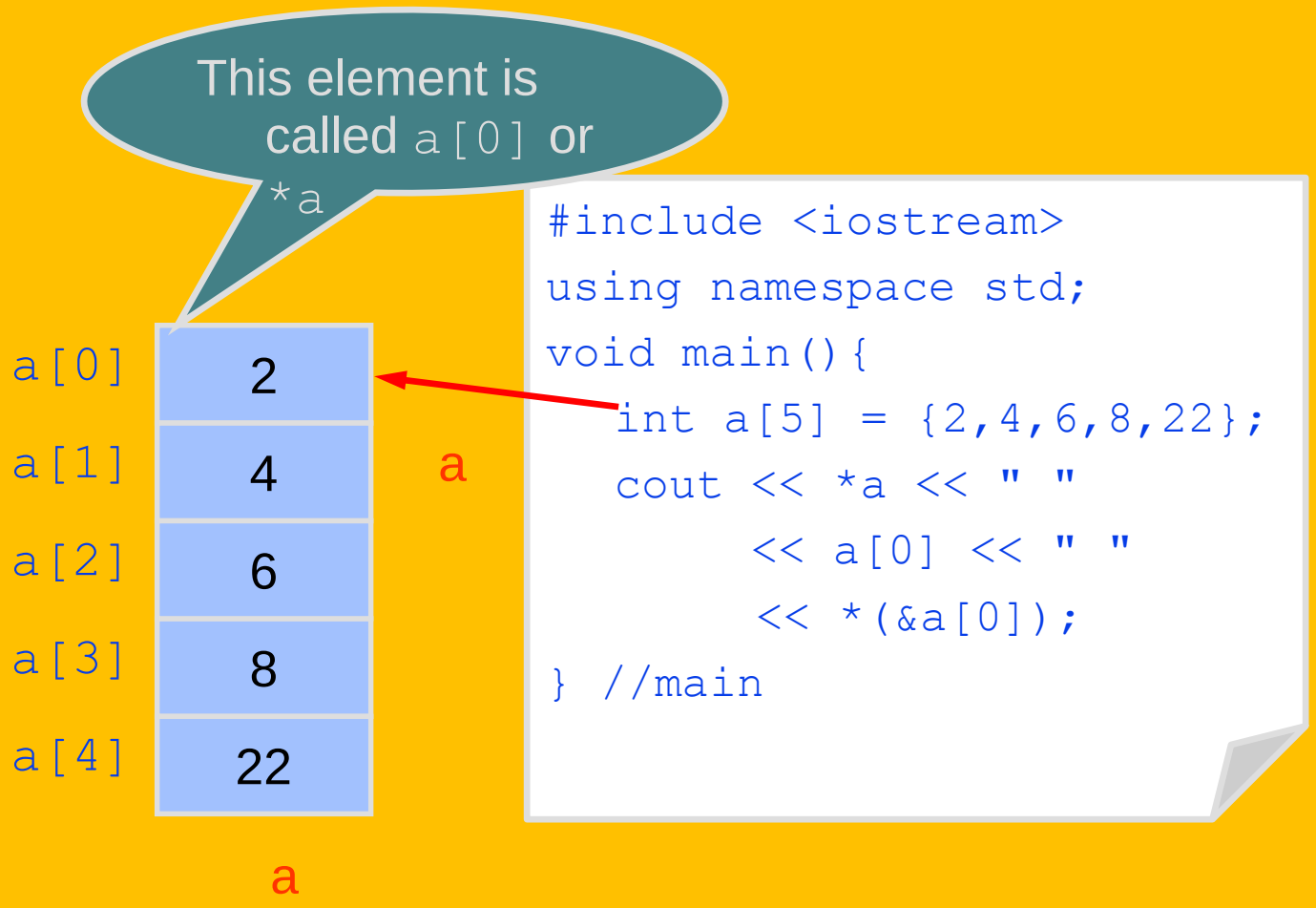
- The name of an array refers only to the address of the first element not the whole array.



Array Name is a Pointer Constant

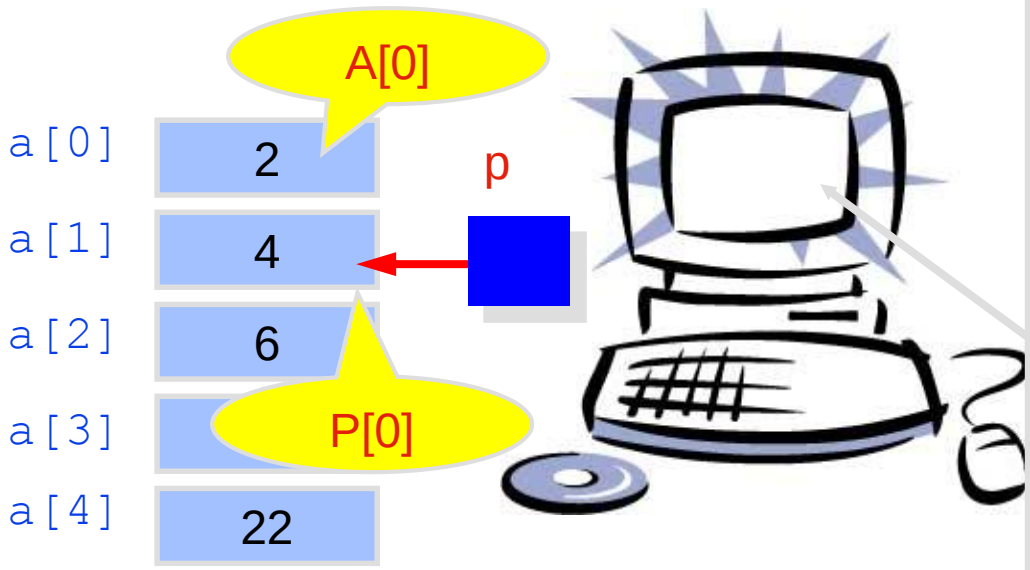
```
#include <iostream>
using namespace std;

int main (){
       // Demonstrate array name is a pointer constant
       int a[5];
       cout << "Address of a[0]: " << &a[0] << endl
            << "Name as pointer: " << a << endl;
       return 1;
}

/* result:
Address of a[0]: 0x0065FDE4
Name as pointer: 0x0065FDE4
*/
```

Dereference of An Array Name



Multiple Array Pointers

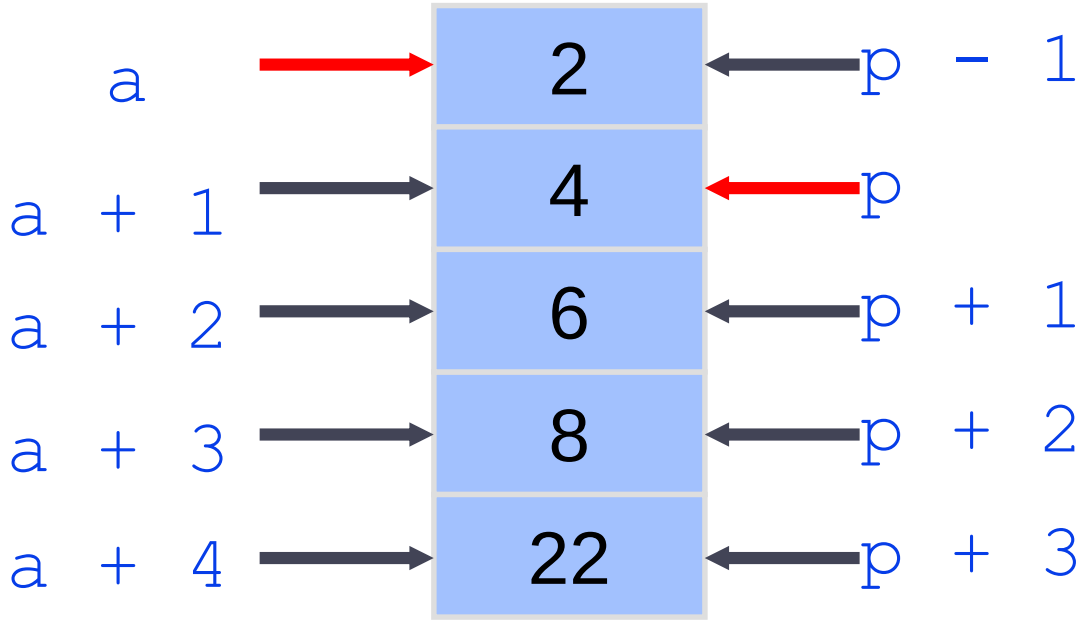
- Both `a` and `p` are pointers to the same array.



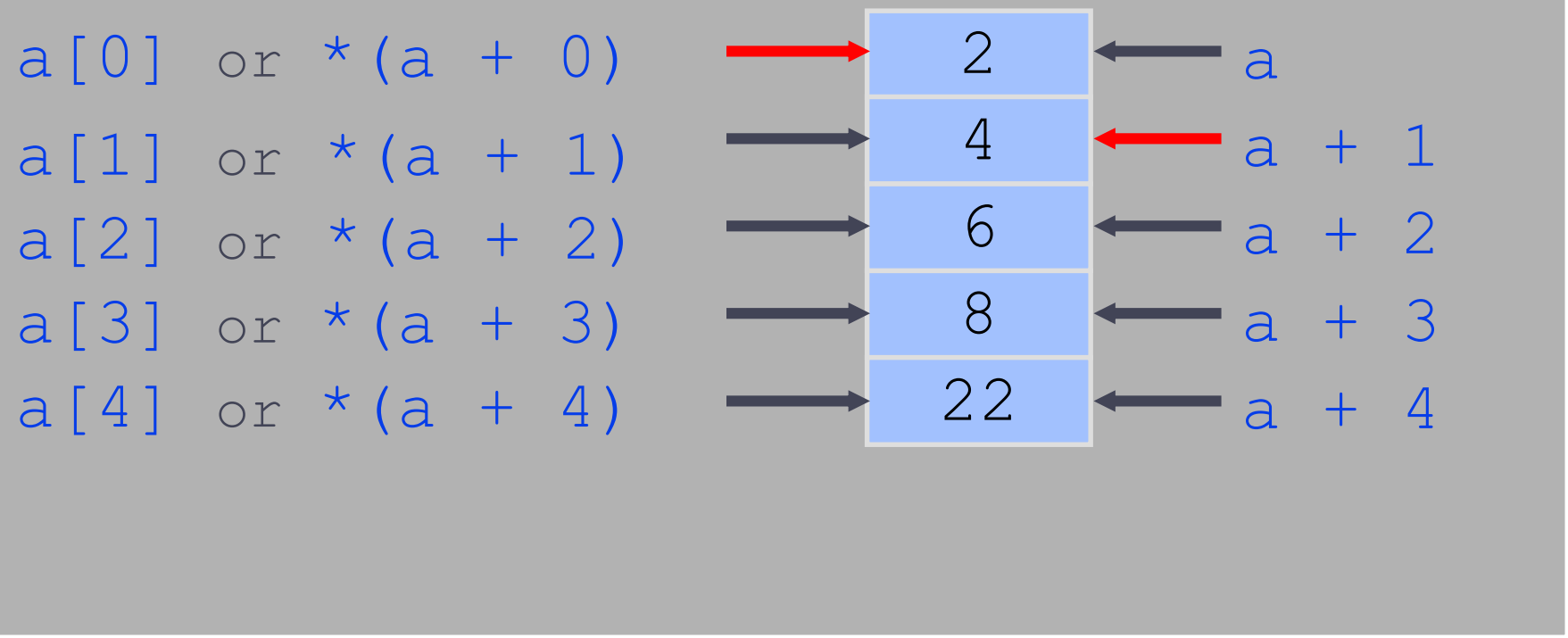

```
#include <iostream>
using namespace std;
int main(){
   int a[5] = {2,4,6,8,22};
   int *p = &a[1];
   cout << a[0] << " "
        << p[-1];
   cout << a[1] << " "
        << p[0];
    return 1;
}
```

Pointer Arithmetic

- Given a pointer `p`, `p+n` refers to the nth element, i.e., offset from `p` by `n` positions.

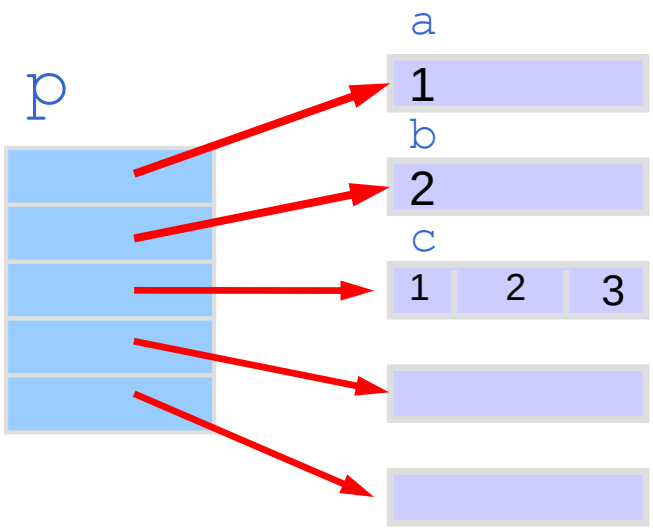


Dereferencing Array Pointers



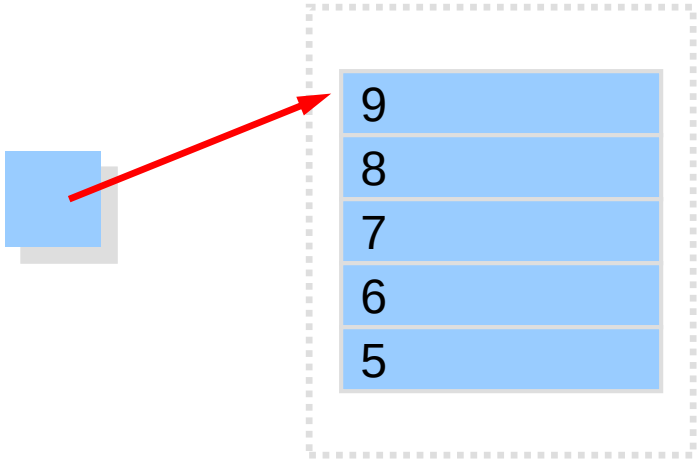
`*(a+n)` is identical to `a[n]`

Array of Pointers & Pointers to Array



An array of Pointers

```
int a = 1, b = 2, c[] =
   {1,2,3};
int *p[5]; // 2D array
p[0] = &a;
p[1] = &b;
p[2] = c;
```



A pointer to an array

```
int list[5] = {9, 8, 7, 6, 5};
int *P;
P = list;//points to 1st entry
P = &list[0];//points to 1st entry
P = &list[1];//points to 2nd entry
P = list + 1; //points to 2nd entry
```

The 2D table and table[0] are of the same address

```
int table[2][2] = {{0,1}, {1,2}};

cout << table << endl;
cout << *table << endl;  //same as above
cout << table[0] << endl; // same as above
cout << *table[0] << endl;
cout << table[0][0] << endl; // same as above
cout << **table << end;  // same as above
```

Output:

```
0xffbff938
0xffbff938
0xffbff938
0
0
0
```

NULL pointer

- NULL is a special value that indicates an empty pointer
- If you try to access a NULL pointer, you will get an error

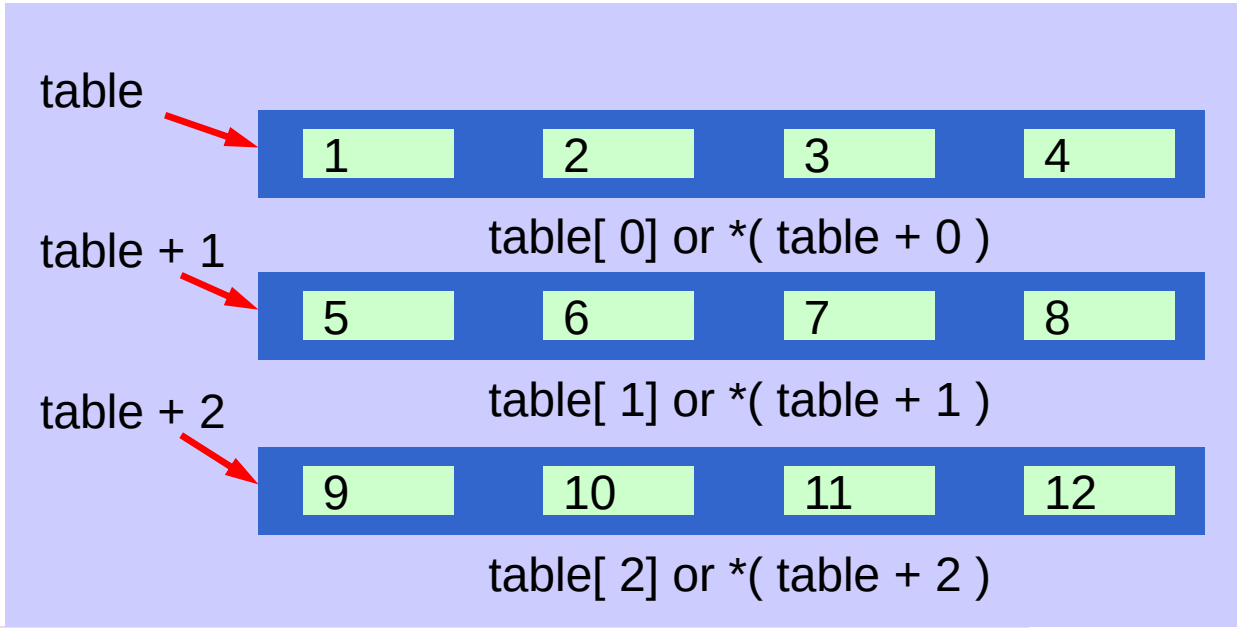
```
int *p;
p = 0;
cout << p << endl; //prints 0
cout << &p << endl;//prints address of p
cout << *p << endl;//Error!
```

Storing 2D Array in 1D Array by Linearization

```
int twod[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
int oned[12];
for(int i=0; i<3; i++){
    for(int j=0; j<4 ; j++)
        oned[i*4+j] = twod[i][j];
}
```

Pointer to 2-Dimensional Arrays

```
table+i =
*(table+i) =
table[i] =
&table[i][0]
refers to
the address
of the ith
row
```



What is
`**table ?`

```
int table[3][4] = {{1,2,3,4},
{5,6,7,8},{9,10,11,12}};

for(int i=0; i<3; i++){
   for(int j=0; j<4; j++)
       cout << *(*(table+i)+j);
   cout << endl;
}
```

`*(table[i]+j) = table[i][j]`

Get the address of the 1st element of the ith row of table, i.e., &table[i][0]
Do NOT put *(table+i+j) because the compiler will mistake it as (i+j) row

`main()`

- Note that `main()` is a function, the parent/mother function of all functions called by the program
  - The operating system first calls/executes this "function"
- Since it is a function, it can have arguments
  - Command line arguments
  - Access it through `argc` and `argv`
  - `argc` is the number of command line arguments
  - `argv` is an array of character pointers (char * *argv)
  - The first argument is always the executable name

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

int main(int argc, char ** argv){

  int i;

  for( i = 0; i < argc; i++ )
    cout << argv[i] << endl;

  return(0);
}
```

Get the number of command-line arguments here

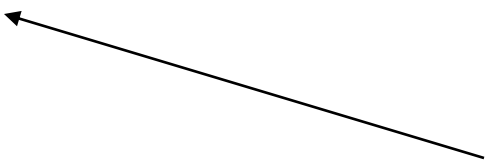
Get the comand strings here, may also be `char * argv[]`

```
cssu5:> a.out 1 2 3
a.out
1
2
3
cssu5:> a.out hello world guys !
a.out
hello
world
guys
!
```

Core Dump

Don't return pointers (or references) to local variables!

```
double * aFunc(void) {
  double d;
  return &d;
}
int main(int argc,
         const char * argv[]) {
  double * pd = aFunc();
  *pd = 3.14;
  return 0;
}
```



Boom!

Dynamic Objects

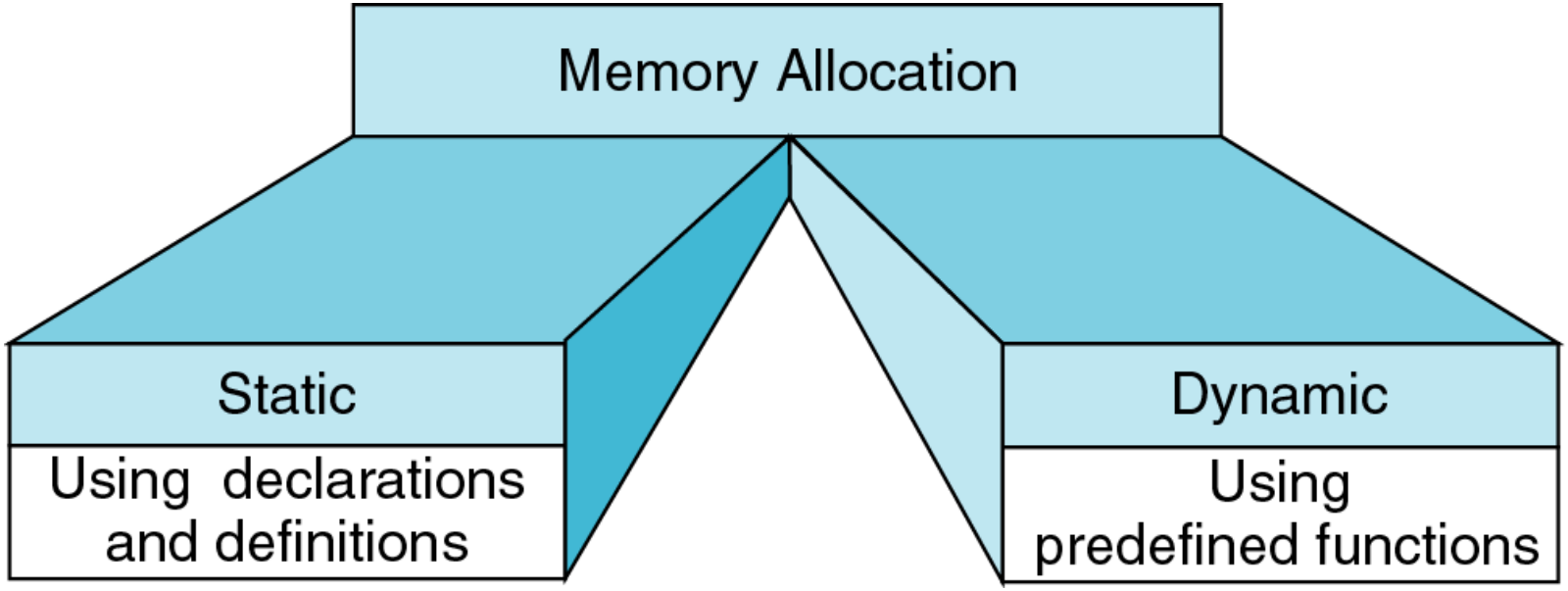
Memory Management

- Static Memory Allocation
  - Memory is allocated at compilation time
  - Allocation on “stack”
- Dynamic Memory
  - Memory is allocated at running time
  - Allocation on “heap”
- Stack usage + Heap usage <= total memory available

Static vs. Dynamic Objects

- Static object
  - Memory is acquired automatically
  - Memory is returned automatically when object goes out of scope
  - E.g., variables as declared in function calls
- Dynamic object
  - Memory is acquired by program with an allocation request
    - `new` operation
  - Dynamic objects can exist beyond the function in which they were allocated
  - Object memory is returned by a deallocation request
    - `delete` operation

Memory Allocation: Stack and Heap



Heap: new delete

```
{
   int a[200];
   …
}
```

```
int* ptr;
ptr = new int[200];
…
delete [] ptr;
```

Object (variable) creation: `New`

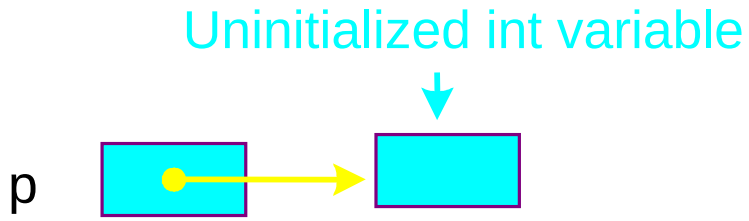
Syntax

```
ptr = new SomeType;
```

where `ptr` is a pointer of type `SomeType`

Example

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



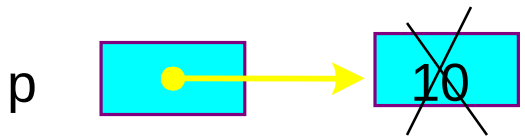
Object (variable) destruction: `Delete`

Syntax

```
delete p;
storage pointed to by p is returned to free store
  and p is now undefined
```

Example

```
int* p = new int;
*p = 10;
delete p;
```



New:Creating dynamic arrays

- Syntax

```
P = new SomeType[Expression];
```

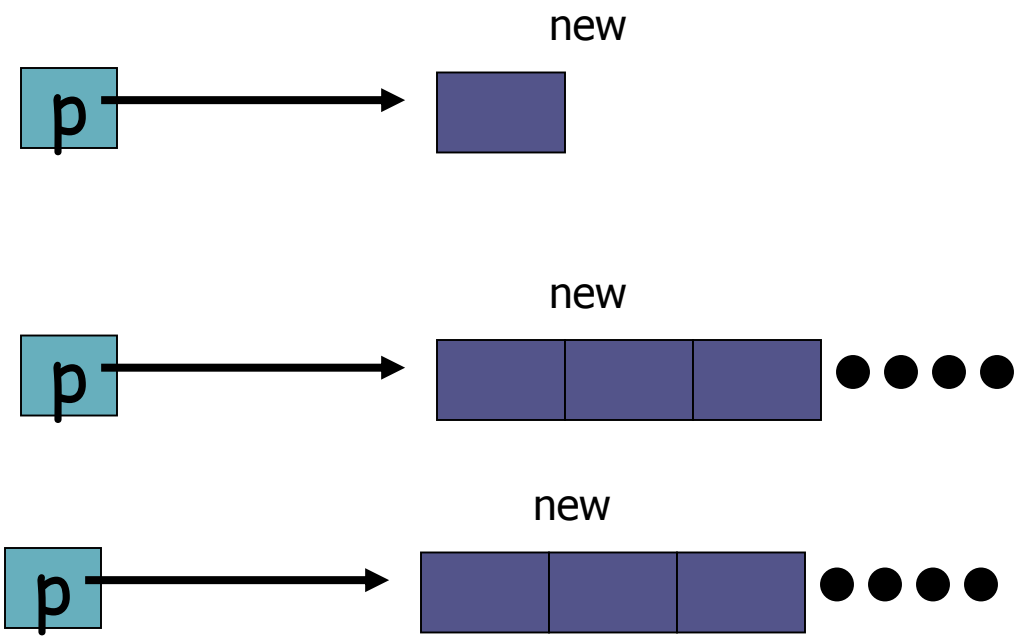
  - Where
    - `P` is a pointer of type `SomeType`
    - `Expression` is the number of objects to be constructed -- we are making an array
- Because of the flexible pointer syntax, `P` can be considered to be an array

An Example

Dynamic Memory Allocation

- Request for "unnamed" memory from the Operating System

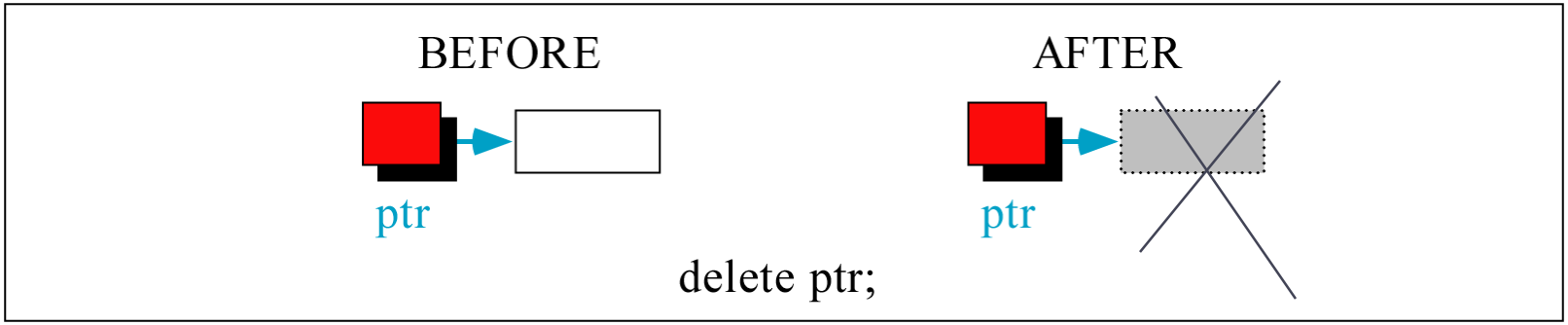
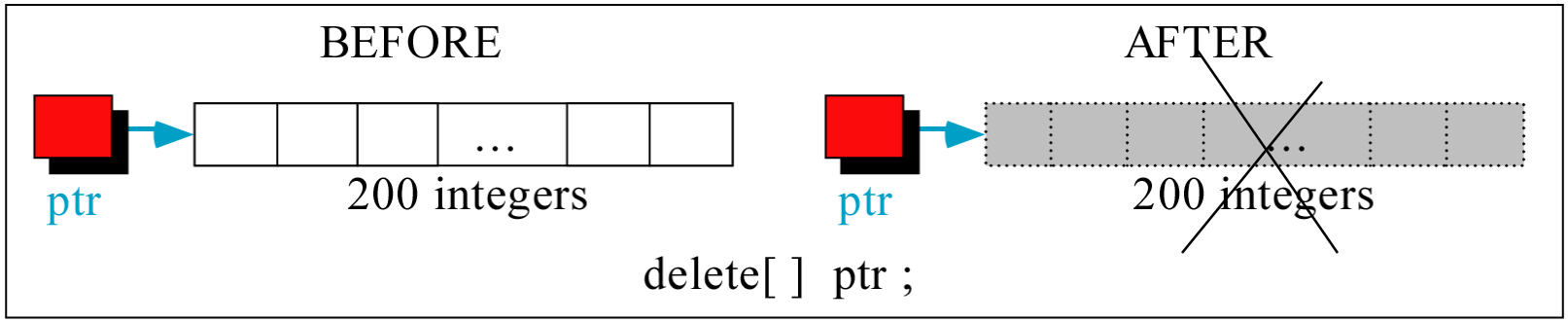
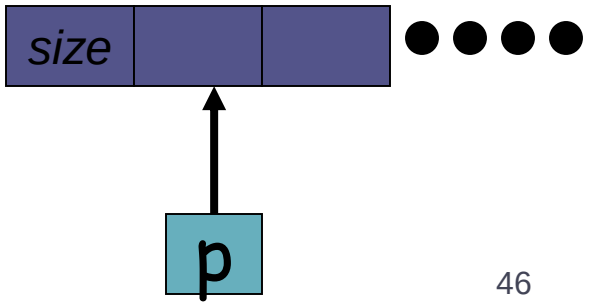
- ```
  int *p, n=10;
  p = new int;

  p = new int[100];
  ```




```
p = new int[n];
```
```

Memory Allocation Example

Need an array of unknown size

```
main()
{
      cout <<  "How many students? ";
      cin   >> n;

      int *grades = new int[n];

      for(int i=0; i < n; i++){
          int mark;
          cout << "Input Grade for Student" << (i+1)  << " ? :";
          cin >> mark;
          grades[i] = mark;
      }

    . . .
    printMean( grades, n ); // call a function with dynamic array
    . . .
}
```

Freeing (or deleting) Memory





How Does C++ Keep Track of the Size of the Array?

- For each object allocated on the heap, there is a leader indicating the size of the array
  - The size of the leader is system dependent (usually 4 bytes)
  - The pointer points to the first useful element of the array
- At de-allocation (`delete [] p`), the system peeps into the leader and releases the array, including the leader



At deletion, p must be at the beginning of the array

- Note that at de-allocation, the system will load the size immediately before the pointer p
  - Therefore, remember to position the pointer to the beginning of the array before deletion
  - Otherwise, the size will be loaded wrongly
- The following is hence bad programming style which does not lead to portable codes:
  - `delete [] (p+1);`
  - `delete p[3];  //delete an element`
  - `p++; delete [] p;`
  - etc.
- Always de-allocate the whole array, not the partial one
- Always de-allocate array on heap; no need to de-allocate variables on stack

Can I delete a NULL pointer?

- It is ok to (repeatedly) delete a NULL pointer. It does nothing. Therefore, it is not necessary to check whether a pointer is NULL before deletion.
  - if (p != NULL) delete p; // if condition is not needed
- It is, however, an error to *repeatedly* delete a non-null pointer:

```
int * p = new int[10];
delete [] p;
delete [] p; // compilation error: double free error
```

- Therefore, it is always a good practice to set a pointer to somewhere valid (such as NULL) *after* its deletion:

```
int * p = new int[10];
delete [] p;
p = NULL; // or somewhere valid, e.g., p = new double[5];
```

- Note that `new[]` must be paired with `delete []`, because they may work differently as compared with `new` and `delete` (without a squared bracket):

```
int * iptr = new int[100];
delete [] iptr; // should not delete iptr;
```

A Simple Dynamic List Example

```
cout << "Enter list size: ";
int n;
cin >> n;
int *A = new int[n];
if(n<=0){
   cout << "bad size" << endl;
   return 0;
}
initialize(A, n, 0);  // initialize the array A with value 0
print(A, n);
A = addElement(A,n,5);  //add an element of value 5 at the end of A
print(A, n);
A = deleteFirst(A,n);  // delete the first element of A and
                // assign the new array back to A (Clumsy statement)
print(A, n);
selectionSort(A, n);  // sort the array (not shown)
print(A, n);
delete [] A;
```

Initialize

```
void initialize(int list[], int size, int value){
  for(int i=0; i<size; i++)
      list[i] = value;

}
```

print()

```
void print(int list[], int size) {
   cout << "[ ";
   for(int i=0; i<size; i++)
        cout << list[i] << " ";
   cout << "]" << endl;
}
```

Delete the first element

```
// for deleting the first element of the array
int* deleteFirst(int list[], int& size){
  if(size <= 1){
       if( size) delete list;
       size = 0;
       return NULL;
  }
  int* newList = new int [size-1]; // make new array
  if(newList==0){
       cout << "Memory allocation error for deleteFirst!" << endl;
       exit(0);
  }
  for(int i=0; i<size-1; i++)     // copy and delete old array
       newList[i] = list[i+1];
  delete [] list;
  size--;
  return newList;
}
```

Adding Elements

```
// for adding a new element to the end of array
// return the newly created array; list array is destroyed
int* addElement(int list[], int& size, int value){
   int* newList = new int [size+1]; // make new array
   if(newList==0){
        cout << "Memory allocation error for addElement!" << endl;
        exit(0);
   }
   for(int i=0; i<size; i++)
        newList[i] = list[i];
   if(size) delete [] list;
   newList[size] = value;
   size++;
   return newList;
}
```

To alter the original list, one may call

```
list = addElement( list, size, 100 );
```

Would like to replace it by:

```
addElement( list, size, 100);
```

Main program

```
int main(){

  int * A = NULL;
  int size = 0;
  int i;

  for( i = 0; i < 10; i++ )
    addElement( A, size, i );

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

  for( i = 0; i < 4; i++ )
    deleteFirst( A, size );

  for( i = 0; i < 6; i++ )
    cout << A[i] << " ";
  cout << endl;

  return 0;
}
```

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

Adding Element (version 2): Transparent Alteration on `list`

```
// for adding a new element to end of array
// list and size are altered directly
void addElement( int * & list, int & size, const int value ){

  int * newList = new int [size + 1];

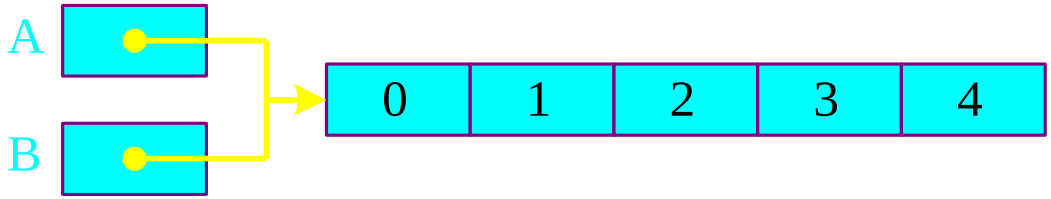
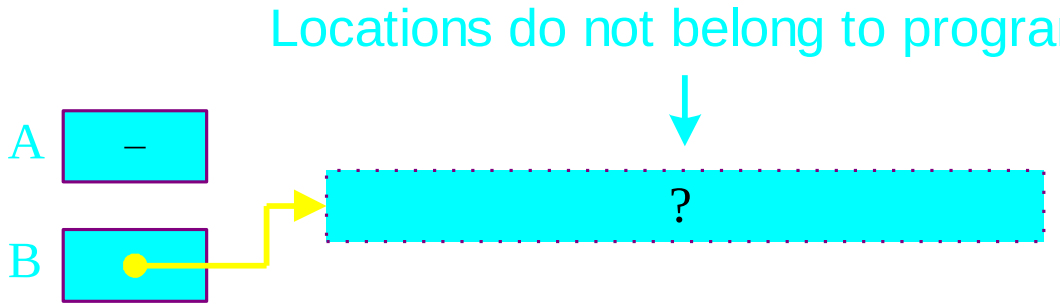
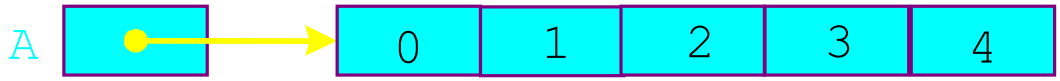
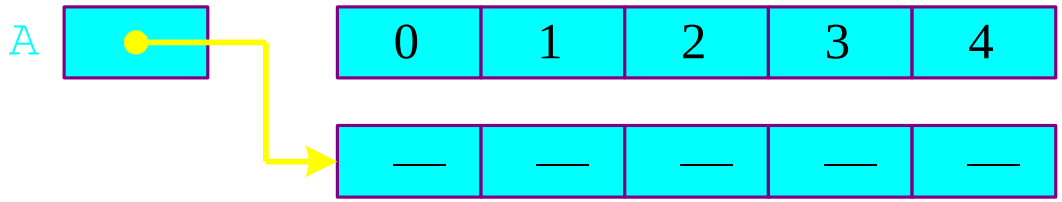
  if( newList == NULL ){
    cout << "Memory allocation error for addElement!" << endl;
    exit(-1);
  }

  for( int i = 0; i < size; i++ )
    newList[ i ] = list[ i ];  // copy over

  if( size )
    delete [] list;

  newList[ size ] = value; // last element takes value
  size++;
  list = newList;   // this is newly added

  return;
}
```

Deleting Element (version 2)

```
void deleteFirst( int * & list, int & size ){

  //same as before
  ...
  list = newList;
  return;
}
```

Dangling Pointer Problem

```
int *A = new int[5];
for(int i=0; i<5; i++)
    A[i] = i;
int *B = A;
```




```
delete [] A;
B[0] = 1; // illegal! Segmentation fault
```



Memory Leak Problem: Heap memory which is impossible to be accessed again

```
int *A = new int [5];
for(int i=0; i<5; i++)
    A[i] = i;
```




```
A = new int [5];
```

These locations cannot be accessed by program



Another Leak Example: Returning a dereferenced pointer

- After foo returns the value of the integer, the memory allocated to iptr can no longer be accessed.
- How can we fix it?
  - Deallocating the memory before you exit by copying the value to a local integer first
  - Returning the pointer so as to pass the pointer responsibility to the caller

```
#include <iostream>

using namespace std;

int foo(){
  int * iptr = new int;
  *iptr = 10;
  return *iptr;
}

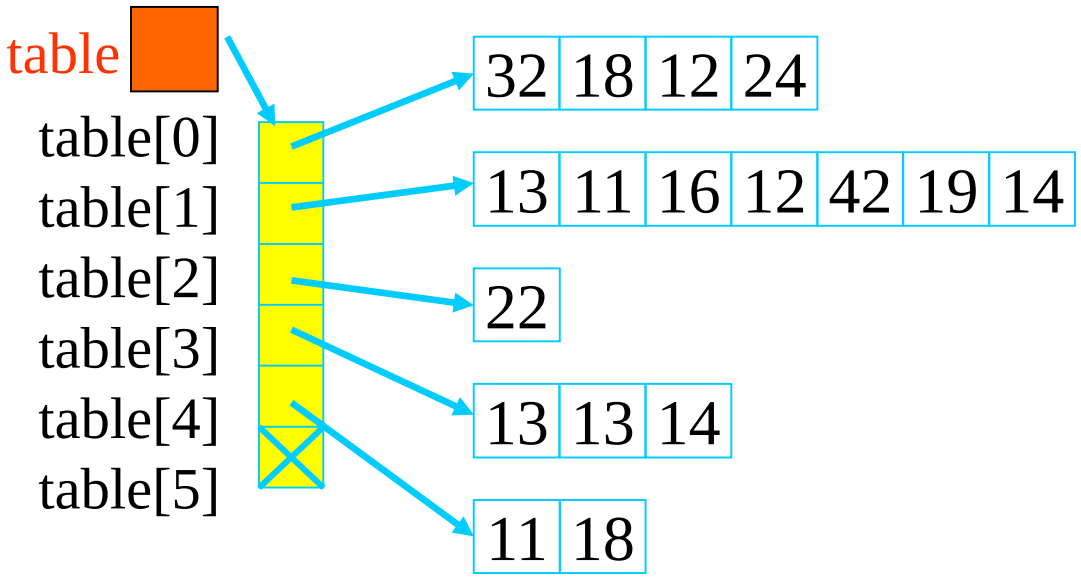
int main(){
  int i = foo();
  //…
}
```

Memory Leak

- Memory leak is only for the program execution time
  - When your main exits, all the memory allocated in the heap will be relinquished by the operating system
  - Even so, there are many programs which are not supposed to exit (server program, Window OS, monitoring programs, etc)
- Therefore, manage your memory carefully
  - De-allocate your memory whenever the objects are no longer needed
- Leak is a SERIOUS bug, even though it is hard to be tested and traced
  - Usually indicated by ever-increasing memory requirement with the execution time of the program

A Dynamic 2D Array

- A dynamic 2D array is an array of pointers to save space when not all rows of the array are full.
- `int **table;`




```
table = new int*[6];
…
table[0] = new int[4];
table[1] = new int[7];
table[2] = new int[1];
table[3] = new int[3];
table[4] = new int[2];
table[5] = NULL;
```

Memory Allocation

```
int **table;

table = new int*[6];

table[0]= new int[3];
table[1]= new int[1];
table[2]= new int[5];
table[3]= new int[10];
table[4]= new int[2];
table[5]= new int[6];

table[0][0] = 1; table[0][1] = 2; table[0][2] = 3;
table[1][0] = 4;
table[2][0] = 5; table[2][1] = 6; table[2][2] = 7;
table[2][3] = 8; table[2][4] = 9;

table[4][0] = 10; table[4][1] = 11;
cout << table[2][5] << endl;
```

Memory Deallocation

- Memory leak is a serious bug!
- Each row must be deleted individually
- Be careful to delete each row *before* deleting the table pointer.
  - ```
    for(int i=0; i<6; i++)
            delete [ ] table[i];
    delete [ ] table;
    ```
    ```

Creating a 2D matrix of size m by n

```
int m, n;
cin >> m >> n >> endl;
int** mat;
mat = imatrix(m,n);
mat[1][3] = 8;
…

int** imatrix(int nr, int nc) {
   int** m;
   m = new int*[nr];
   for (int i=0;i<nr;i++)
       m[i] = new int[nc];
   return m;
}
```

Constant pointer and constant object

- `const int * iptr` means that the object being pointed to by iptr is constant and cannot be changed
  - This is the same as `int const * iptr`
  - You can NEVER do `*iptr = 10;`
  - However, you may do reassignment `iptr = bptr;`
- `int * const iptr` means that the pointer is a constant, not the object that it points to
  - You have to initialize the pointer by `int * const iptr = &a;`
  - Can NEVER have `iptr = &b;`
  - Can have `*iptr = 4;`

const

```
#include <iostream>

using namespace std;

// illustration of static variable
// same as void foo1( int const ** bar )
void foo1( const int ** bar ){

  bar[1][1] = 5;
// change int invalid: Compiler complains
  bar[1] = new int[3];  // change int *
  return;
}

void foo2( int * const * bar ){

  bar[1][1] = 5;  // change int
  bar[1] = new int[3];
//change int * invalid:Compiler complains
  bar = new int * [3];  // change int **
  return;
}
```

```
void foo3( int ** const bar ){

  bar[1][1] = 5;  // change int
  bar[1] = new int[3];
// change int *
  bar = new int * [3];
// change int ** -- invalid: Compiler
// complains
  return;
}
int main(){
  int ** iptr;
  int i, j;

  iptr = new int * [10];
  for( i = 0; i < 10; i++ )
    iptr[ i ] = new int [10];
  for( i = 0; i < 10; i++ )
    for( j = 0; i < 10; i++ )
      iptr[i][j] = i*j;

  foo1( iptr );
  foo2( iptr );
  foo3( iptr );
  return 0;
}
```

struct

Motivation

Remember that an array is a collection of variables of same type, a collection of variables of different types is a 'structure'.

- Structures hold data that belong **together**.
- Examples:
  - Student record
    - student id, name, major, gender, start year, …
  - Bank account:
    - account number, name, currency, balance, …
  - Address book:
    - name, address, telephone number, …
- In database applications, structures are called records.

‘Date’ example

* A ‘date’ type:
  - Day (integer)
  - Month (integer)
  - Year (integer)
* Example:

```
struct Date {
    int day;
    int month;
    int year;
} ;
```

The new composite type “Date” structure

has 3 members.

```
Date christmas;
```

Define new variable of type ‘Date’

```
christmas.day  =  25;
christmas.month = 12;
christmas.year = 2003;
```

Access to member variables using dot operator

‘struct’ definition

```
struct <struct-type>{
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} ;
```

Each identifier defines a member of the structure.

It is usually a ‘global’ definition!

* Example:

```
struct BankAccount{
    string Name;
    int AcountNo[10];
    double balance;
    Date Birthday;
};
```

The “BankAcount” structure has simple array and structure types as members.

* Example:

```
struct StudentRecord{
    string Name;
    int Id;
    string Dept;
    char gender;
};
```

The “StudentRecord” structure has 4 members.

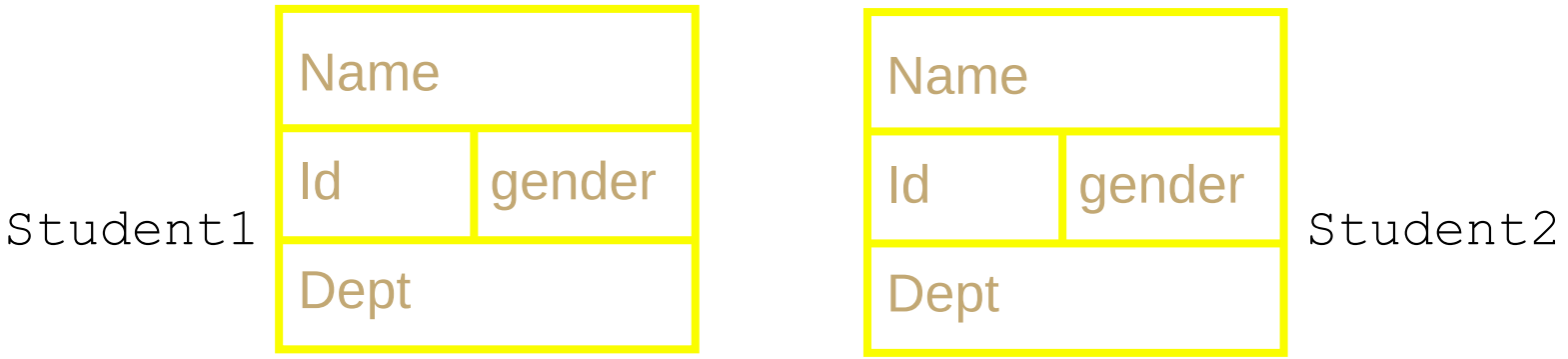
‘struct’ usage

* Declaration of a variable of `struct` type:

  `<struct-type> <identifier_list>;`

* Example:

```
StudentRecord Student1, Student2;
```



`Student1` and `Student2` are variables of `StudentRecord` type.

Member access (dot operator)

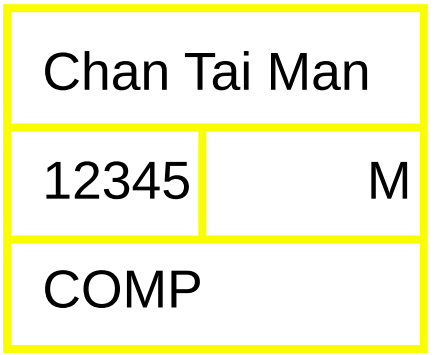
* The members of a struct type variable are accessed with the dot (.) operator:

```
<struct-variable>.<member_name>;
```

* Example:

```
Student1.Name = "Chan Tai Man";
Student1.Id = 12345;
Student1.Dept = "COMP";
Student1.gender = 'M';
cout << "The student is ";
if (Student1.gender = 'F'){
    cout << "Ms. ";
else
    cout << "Mr. ";
}
cout << Student1.Name << endl;
```

Student1



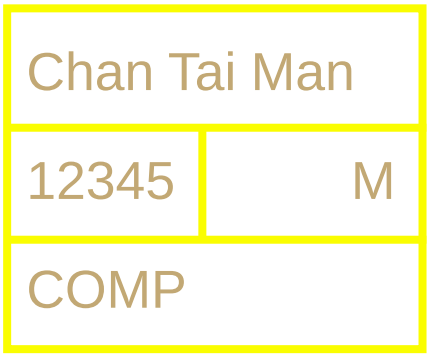
struct-to-struct assignment

* The value of one struct type variable can be assigned to another variable of the same struct type.
* Example:

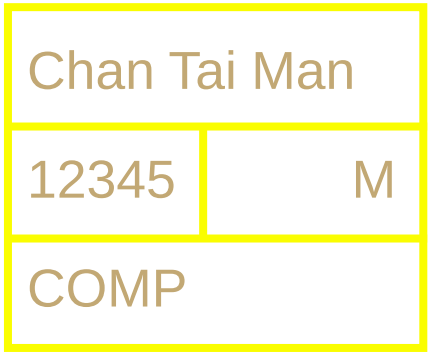
```
Student1.Name = "Chan Tai Man";
Student1.Id = 12345;
Student1.Dept = "COMP";
Student1.gender = 'M';
```

```
Student2 = Student1;
```

Student1

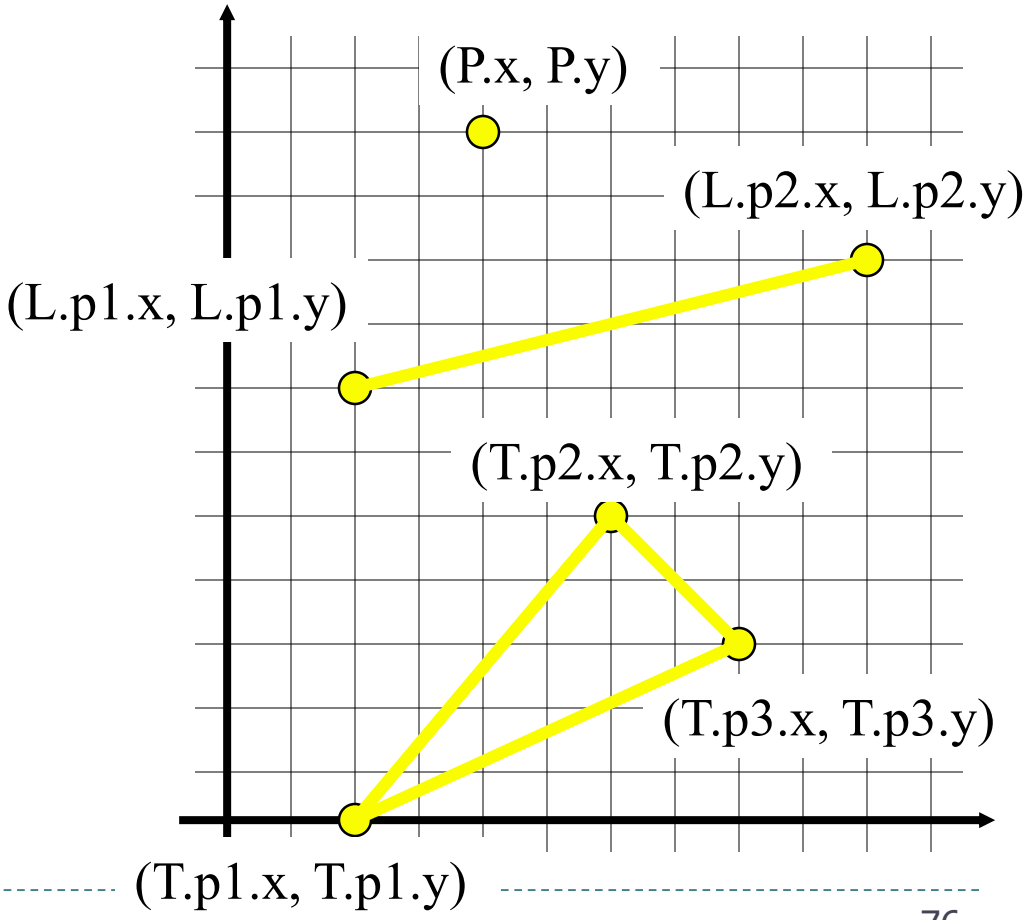


Student2



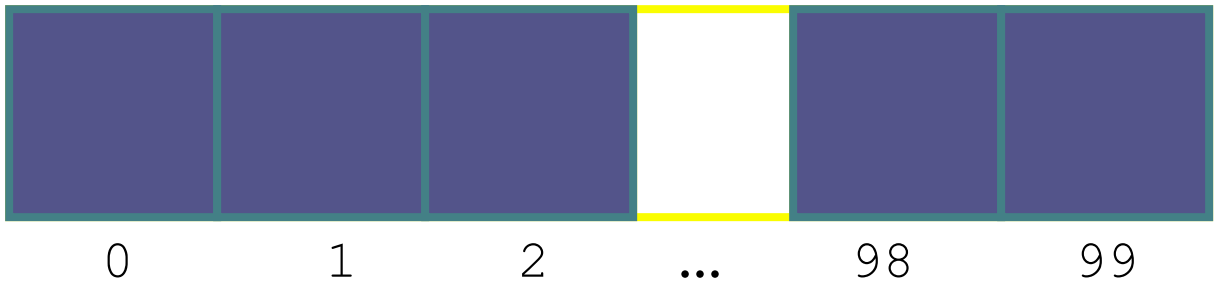
Example of Nested structures

```
struct Point{
    double x, y;
};

struct Line{
    Point p1, p2;
};

struct Triangle{
    Point p1, p2, p3;
};

Point P = {4, 11};
Line L;
Triangle T;
```

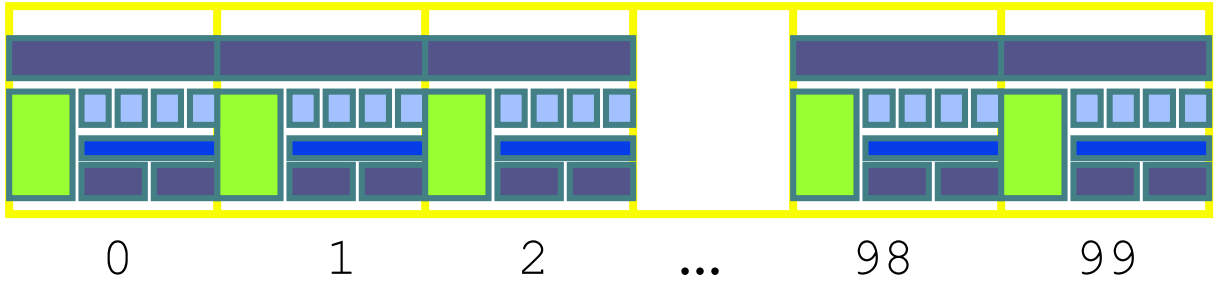


Arrays of structures

* An ordinary array: One type of data

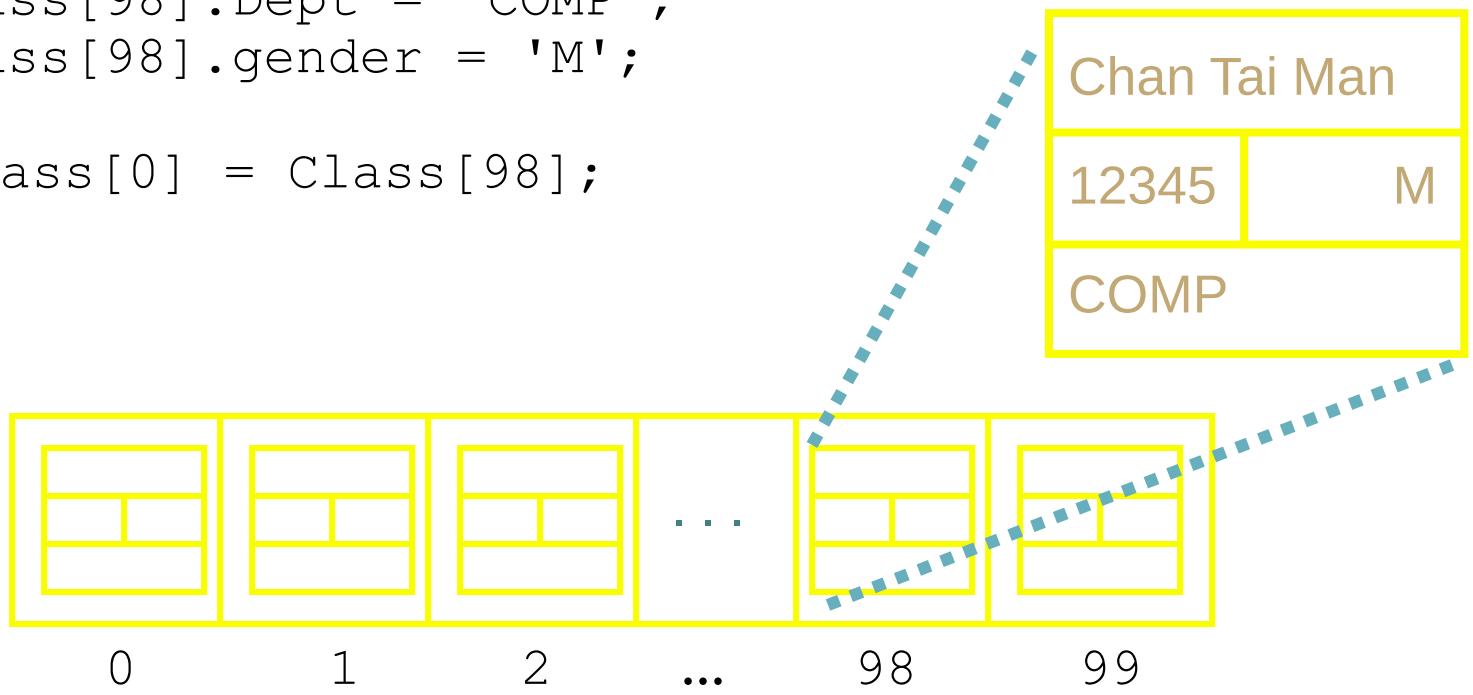


* An array of structs: Multiple types of data in each array element.



* Example:

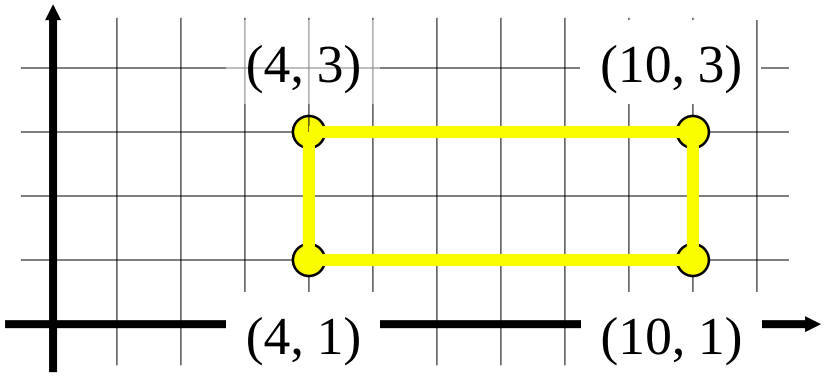
```
StudentRecord Class[100];
Class[98].Name = "Chan Tai Man";
Class[98].Id = 12345;
Class[98].Dept = "COMP";
Class[98].gender = 'M';

 Class[0] = Class[98];
```



Arrays inside structures

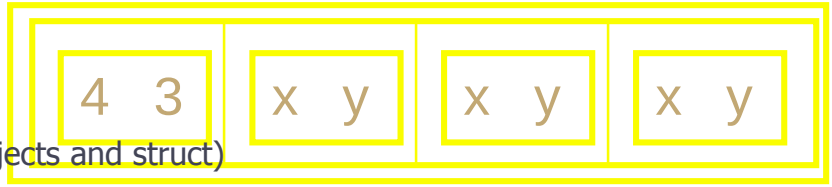
- We can use arrays inside structures.
- Example:

```
struct square{
    point vertex[4];
};
square sq;
```



- Assign values to `Sq` using the given square

```
sq.vertex[0].x = 4;
sq.vertex[0].y = 3;
```



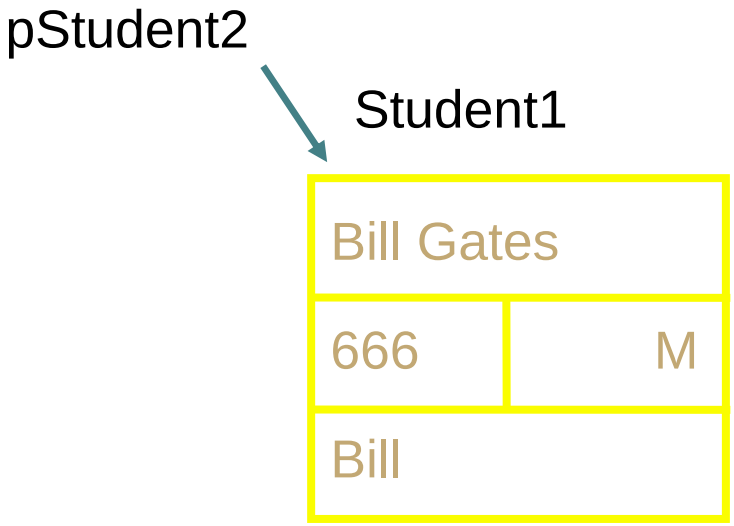
Pointer to Struct

- Declaration of pointer to struct

  <struct-type>* <identifier_list>;

- Example:

```
StudentRecord Student1;
StudentRecord* pStudent2;

Student1.Name = "Bill Gates";
Student1.Id = 666;
Student1.Dept = "Bill";
Student1.gender = 'M';

pStudent2 = &Student1;
(*pStudent2).Id = 444;
pStudent2->Id = 555;
pStudent2->gender = '!';
pStudent2->gender = '?';
```



Dot operator for 'object'

→ for 'pointer'

Struct Initialization and Definition

```
struct children;  // definition prototype (for node)

struct node{
  string str;
  children * cptr;
};

struct children{
  node * nptr;
  children * cptr;
};

int main(){
  node nd = {"Hello", NULL};
  children chld = {&nd, NULL};
. . .
}
```