Pointers

Pointers

- It provides a way of accessing a variable without referring to its name.
- The mechanism used for this is the ***address*** of the variable.

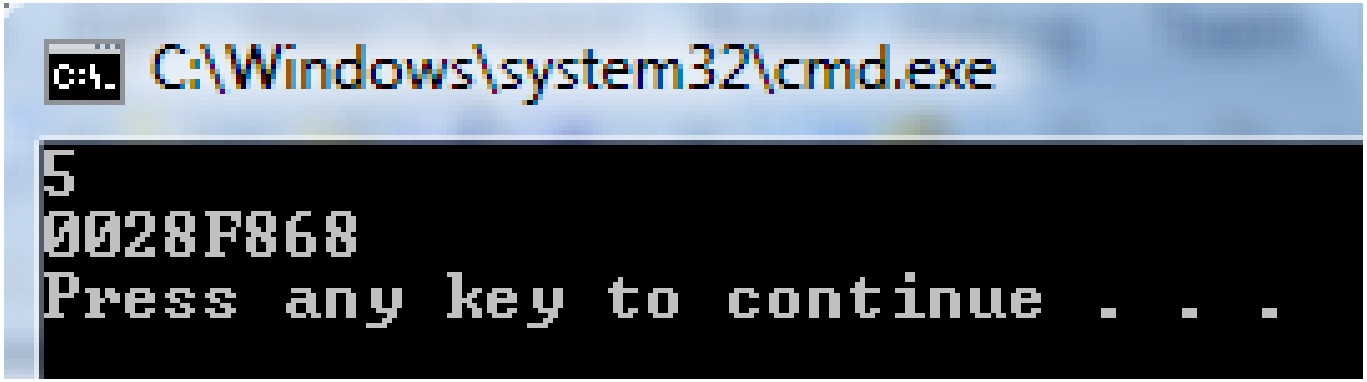
Address of variable

- Each variable occupies some bytes in memory (according to its size)
- Each byte of memory has a unique address so that it can be accessed (just like the address of our homes)
- Variable names are actually titles given to these addresses
  - Such as the ‘white house’
- When we use a variable, the compiler fetches the value placed at its address or writes a new value to that address

Address of variable

- How can we find the address of that variable?
- The “&” (Ampersand) operator returns the address of a variable

```
cout << a << endl;
cout << &a << endl;
```



Pointers

- Pointers are just variables storing numbers – but those numbers are memory addresses
- A pointer that stores the address of some variable x is said to ***point to*** **x**.
- A pointer is declared by putting a star (or '*') before the variable name.
- To access the **value** of x, we'd need to *dereference* the pointer.

Motivation for using Pointers

- To return more than one value from a function.
- To pass arrays and strings more conveniently from one function to another.
- To manipulate arrays more easily by moving pointers to them.

Motivation for using Pointers

- In more advanced programming, such as
  - To create complex data structures, such as linked lists and binary trees
  - For dynamic memory allocation and de-allocation

Pointers

- Declaration of pointer to an int

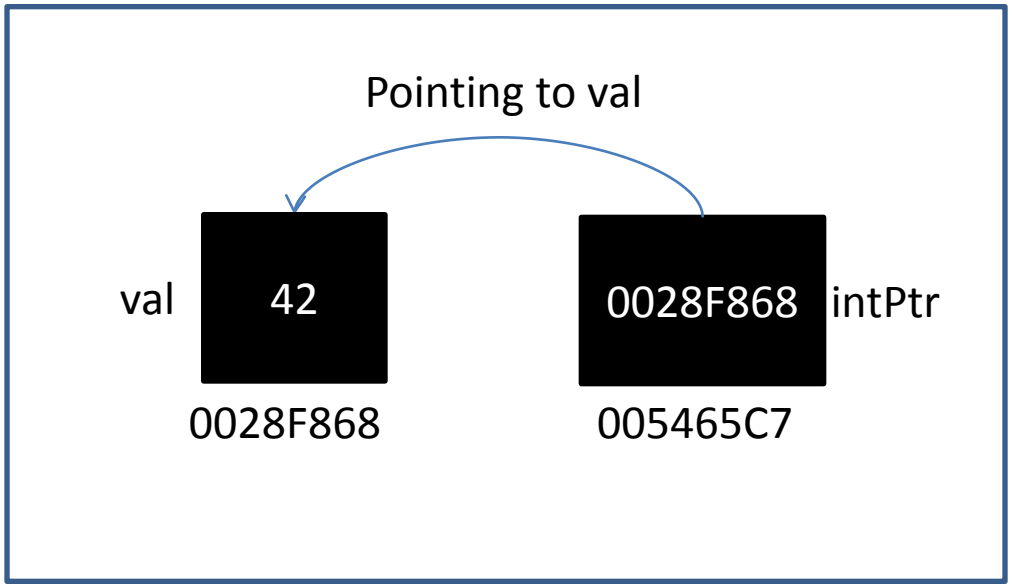
```
int*  intPtr;
```

- Initializing a pointer

```
int val = 42;
int* intPtr = &val;
```

- The ampersand (or '&') before val means "take the address of val".
- The pointer intPtr is assigned the address of val, which is another way of saying that intPtr now points to val.

Pointers



Pointers

- If we print out the value of intPtr, we'll get the address of val.
- To get the data that intPtr actually points at, we have to *dereference* intPtr with the *unary star operator '*'*

```
int val = 42;
int *intPtr = &val;
cout << "&val: " << &val << endl;//address of val
cout << "intPtr: " << intPtr << endl;   // ..again, address
cout << "*intPtr: " << *intPtr << endl; // displays 42
```

- **Note that the star (or '*') operator has many uses. When used in a binary context, it can mean multiplication. When used in a variable declaration, it means "this variable is a pointer". When used in an expression as a single-argument operator, it can mean "dereference". With so many uses, it's easy to get confused.**

Pointers

- Since intPtr points to val, any changes that we make to val will also show up when we dereference intPtr:

```
int val = 42;
int* intPtr = &val;
cout << "val: " << val << endl; // displays 42
cout<<"*intPtr: "<<*intPtr<<endl;//displays 42
val = 999;
cout << "val: " << val << endl;// displays 999
cout << "*intPtr: " << *intPtr << endl; //
displays 999
```

Pointers

- You can declare a pointer to any type, not just int:

```
char ch= ‘H’;
char* chPtr = &ch;
cout << “ch: " << ch<< endl;
cout << “chPtr: " << chPtr << endl;
cout << "*chPtr: " << *chPtr << endl;
*chPtr = "!";
cout << "*chPtr: " << *chPtr << endl;
cout << “ch: " <<ch<< endl;
```

Pointers

```
float val = 1.43;
float* fltPtr = &val;
cout << “val: " << val << endl;          // 1.43
cout << “val: " << fltPtr<< endl;        // address
cout << "*fltPtr: " << *fltPtr << endl;  // 1.43
*fltPtr = 3.1416;
cout << "*fltPtr: " << *fltPtr << endl;   //3.1416
cout << “val: " << val << endl;          // 3.1416
```

Pointers and **const**

- Pointers can be declared **const** in three ways:
  - The pointer itself can be declared **const**
  - The data the point points (“the pointee”) can be declared **const**
  - Both the pointer and the pointee can declared **const**

Pointers and **const**

– **const** **p**ointer, which means that once initialized it can't point at something else.

```
int val1 = 42;
int * const intPtr = &val1;
*intPtr = -1;          // okay
int val2 = 999;
intPtr = &val2;        // error!
```

Pointers and **const**

- **const** data, which means that the data that the pointer points to can't be changed

```
int val1 = 42;
const int * intPtr = &val1;
*intPtr = -1;          // error!
int val2 = 999;
intPtr = &val2;        // okay
```

Pointers and **const**

– Both of the above -- you can change neither the pointer nor the data it points to:

```
int val1 = 42;
const int * const intPtr = &val1;
*intPtr = -1;        // error!
int val2 = 999;
intPtr = &val2;      // error!
```

Pointers and **const**

– The variety of ways that you can use const can be confusing. One useful way to understand pointer declarations is to read them starting from the variable name reading from right to left .

- `int * intPtr;  //the variable intPtr is a pointer to an int.`

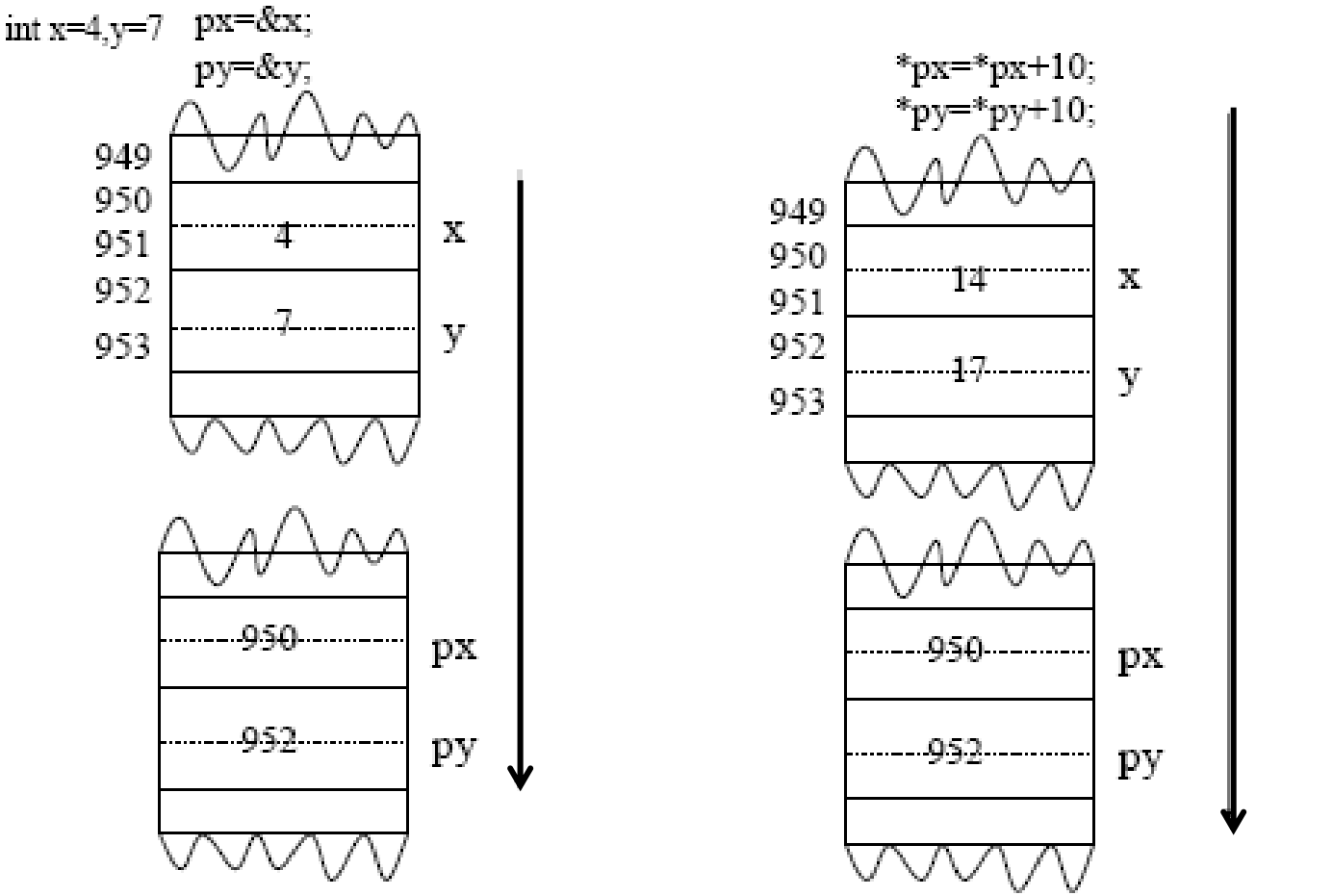
- `const int * intPtr;  //the variable intPtr is a pointer to an int //that is constant. Since the int is constant, we can't change //it. We can change intPtr, however.`

- `int * const intPtr;  //the variable intPtr is a constant pointer //to an int. Since intPtr is constant, we can't change it. //However, we can change the int.`

- `const int * const intPtr; //the variable intPtr is a constant //pointer to an int that is constant. Everything is constant.`

Example

```
int main ( )
  { int x=4, y=7;
    int *px, *py;
    px = &x; //int *px=&x;
    py = &y; //int *py=&y;
    cout << “x = ” << x<<“y =”<<y;
    *px = *px + 10;
    *py = *py + 10;
    cout << “x = ” << x<<“y =”<<y;
    return 0;
  }
```



Operation of the addition using pointers

Exercises

```
int main (void)
   { int a;
     int *ptr1;
     char c;
     char *ptr2;
     int fred;
     ptr2=&c;      //ok. Char pointer init with add of a char
     ptr1=&fred;   //ok. int pointer init with the address of
                   an int
     ptr1=&a ;     //ok. int pointer init with the address of
                   an int
```

Exercises

```
ptr2=&fred; // NO. cannot assign the address of an int to a char
ptr2=&'#'; // NO. cannot take the address of an implicit constant
ptr2=&25;  // NO. cannot take the address of an implicit constant
ptr2=&(a +3); // NO. cannot take the address of an expression
}
```

Exercises

```
int main()
    { int c,a=10;
    int *p1=&a;
    c=*p1; //equivalent the expression "c=a"
    *p1=*p1**p1; //equivalent the expression "a=a*a"
    (*p1)++; //equivalent the expression "a++"
    c=*&a;  //equivalent the expression "c=a"
    }
```

Exercises

```
int a = 3;
int b = 4;
int *pointerToA = &a;
int *pointerToB = &b;
int *p = pointerToA;
p = pointerToB;
cout<< a << b << *p; // Prints 344
```

Pointers and Arrays

- In c++, you can treat pointers and arrays very similarily:

```
int array[ ] = {3,1,4,1,5,9};
int* arrayPtr = array;
cout<<"array[0]: ”    <<array[0] << endl;//displays 3
cout<<"arrayPtr[0]: "<<arrayPtr[0]<<endl;//displays 3
arrayPtr++;
cout<<"arrayPtr[0]: "<<arrayPtr[0]<<endl; // displays 1
array++; // error: arrays are const
```

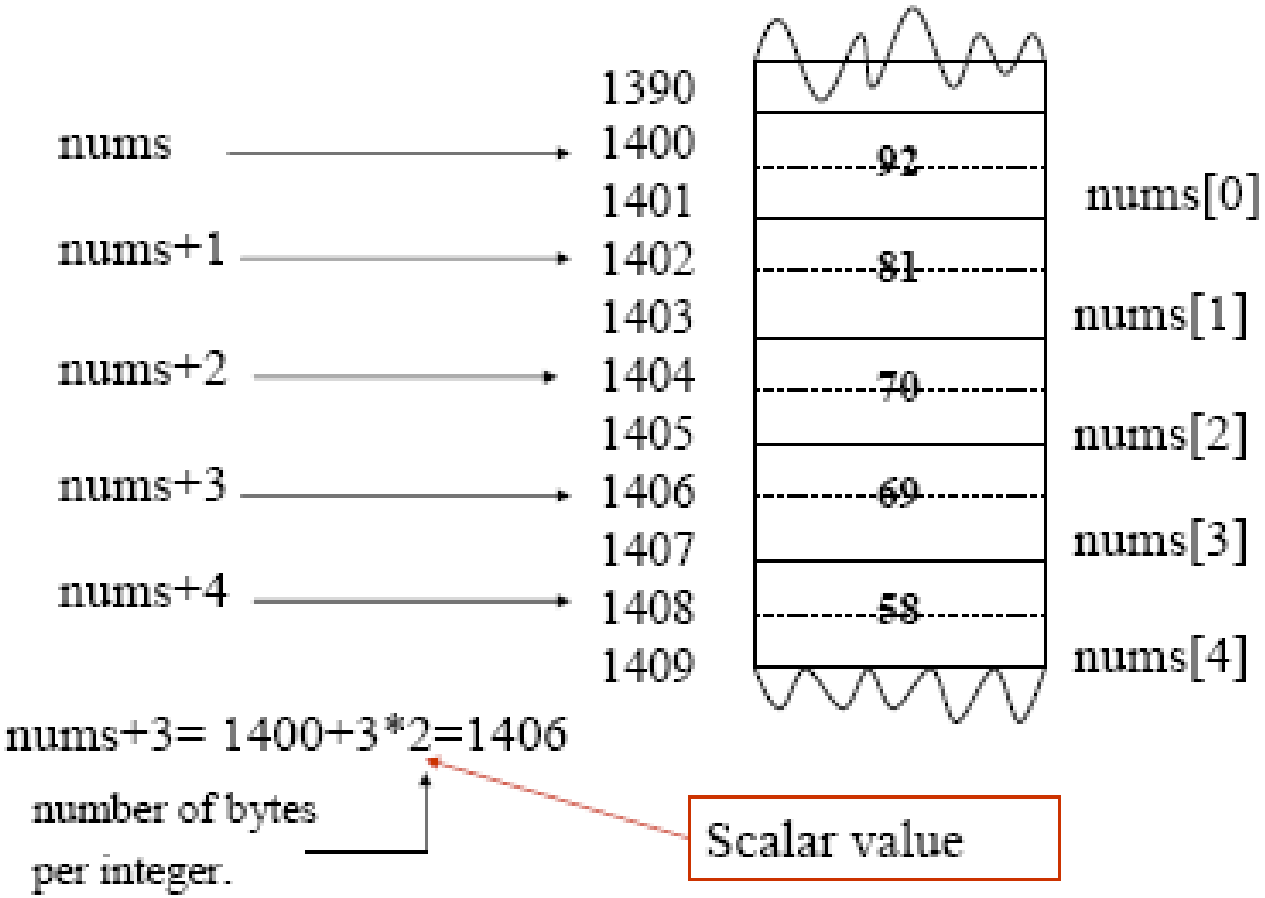
Pointers and Arrays

- Arrays and pointers may be used interchangeably, but you can change what pointers point at, whereas arrays will always "point" to the same thing.

Pointers and Arrays

```
// Using pointers to print out values from
array
int main ( )
  {int nums [5] = {92,81,70,69,58};
   int dex ;
   for ( dex = 0; dex <5; dex++ )
       cout << *(nums + dex)<<“ ”;
  }
```

- ***(nums + dex)** *is same as* **nums [dex]**.


nums
nums+1
nums+2
nums+3
nums+4
1390
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
92
81
70
69
58
nums[0]
nums[1]
nums[2]
nums[3]
nums[4]
nums+3= 1400+3*2=1406
number of bytes
per integer.
Scalar value

Pointers and Arrays

```
int * ptr; /* pointer to int (or int array) */
char *chptr; /*pointer to char(or char array)*/
int table[3]= {5, 6, 7}; /* array */
ptr = table; //&table[0]//*assign array address to pointer*/
cout << *ptr; /*prints 5*/
cout << *(ptr+1); /* prints 6 */
```

Pointers and Functions

```
/* test function that returns two values */
void rets2 (int* , int* ); /*prototype*/
int main ( )
      {int x, y; /* variables*/
       rets2 ( &x, &y ); /*get values from
function*/
       cout <<“First is = ”<< x<<“Second is = ”<<y;
      }
/* returns two numbers */
void rets2 ( int* px, int* py )
      { *px= 3; /* set contents of px to 3 */
        *py = 5; /* set contents of px to 5 */
      }
```

Pointers and Referencing

- Reference and pointers have a more or less similar function. However, there are some fundamental differences in how they work.
- Reference is an alias to the variable, while pointer gives the address of the variable.
- References basically function like pointers that are dereferenced every time they are used.

Pointers and Referencing

- However, because you cannot directly access the memory location stored in a reference (as it is dereferenced every time you use the reference), you cannot change the location to which a reference points, whereas you can change the location to which a pointer points.