CS 1002 Programming Fundamentals
Lecture 29 Nov 2022

Pointers

National University
Of Computer and Emerging Sciences

Overview of Pointers

- Pointer variable (pointer): a variable that holds an address
- Pointers provide an alternate way to access memory locations
- **The * and & operators**
  - & operator is the address operator
  - * operator is the dereferencing operator. It is used in pointers declaration

Needs a Banking Program

User wants a program for making
bank deposits and withdrawals.

(You can write that code by now!)

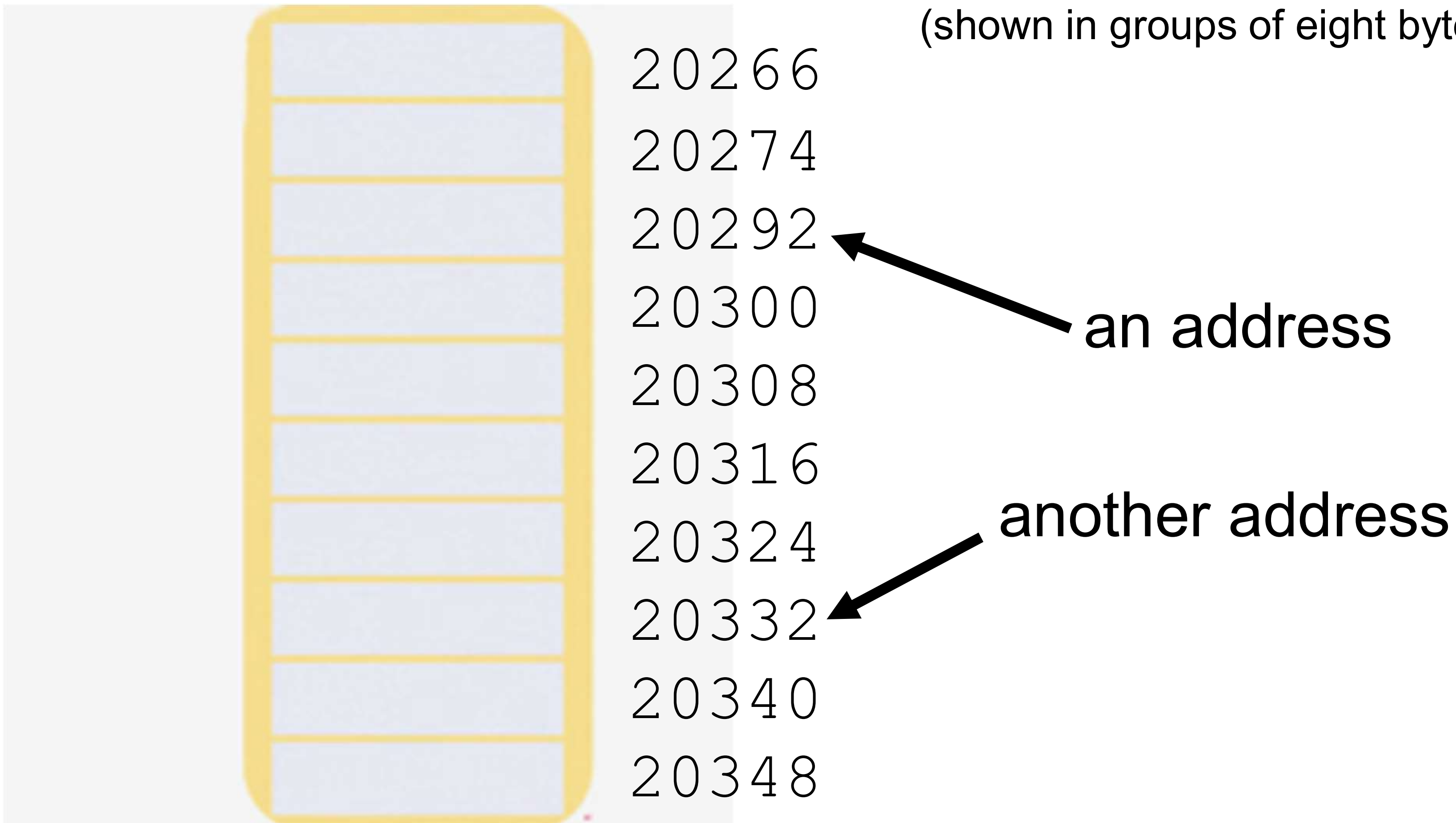
… **`balance += depositAmount`** …
… **`balance -= withdrawalAmount`** …

Addresses and Pointers

Here's a picture of RAM.

Every byte in RAM has an *address*.

(shown in groups of eight bytes)



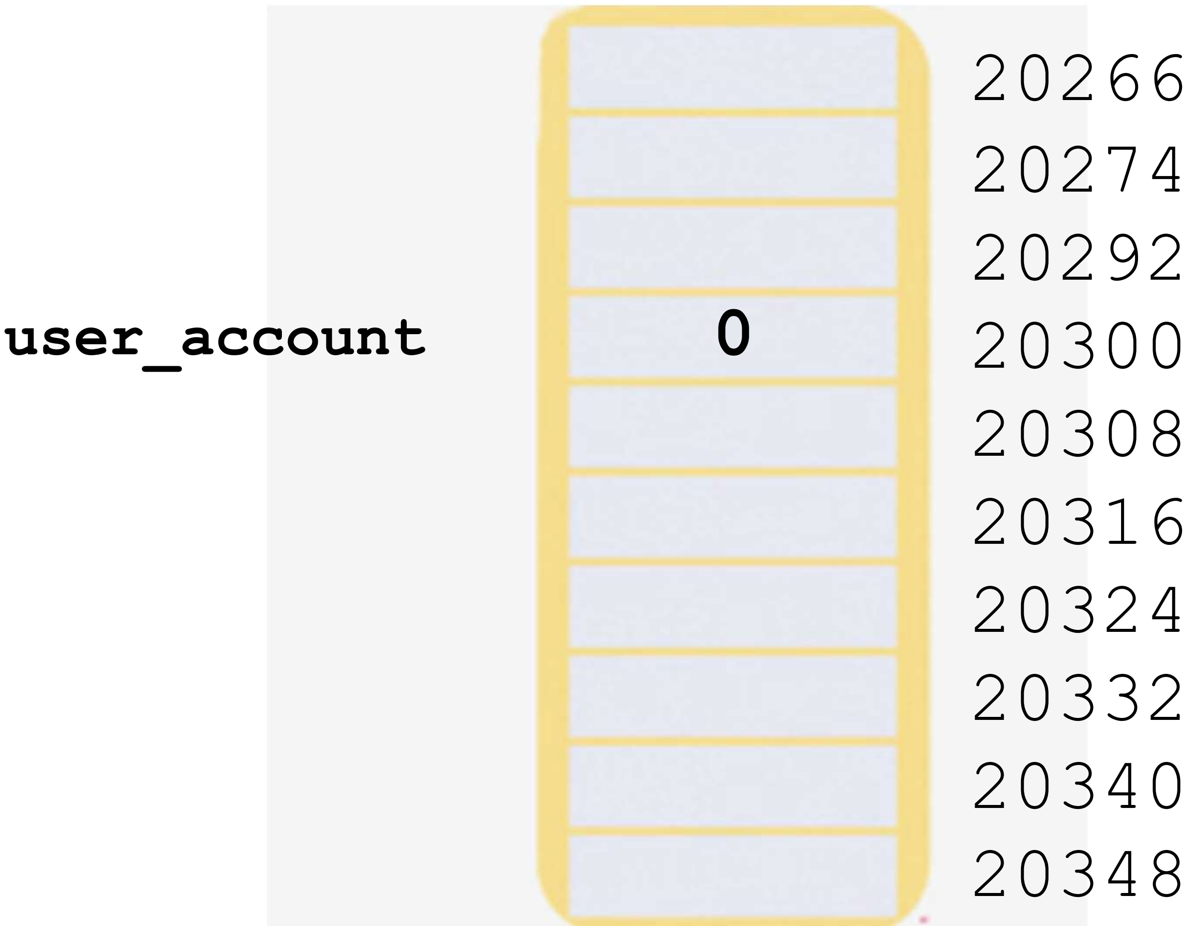
Addresses and Pointers

Here's how we have pictured a variable in the past:

user_account

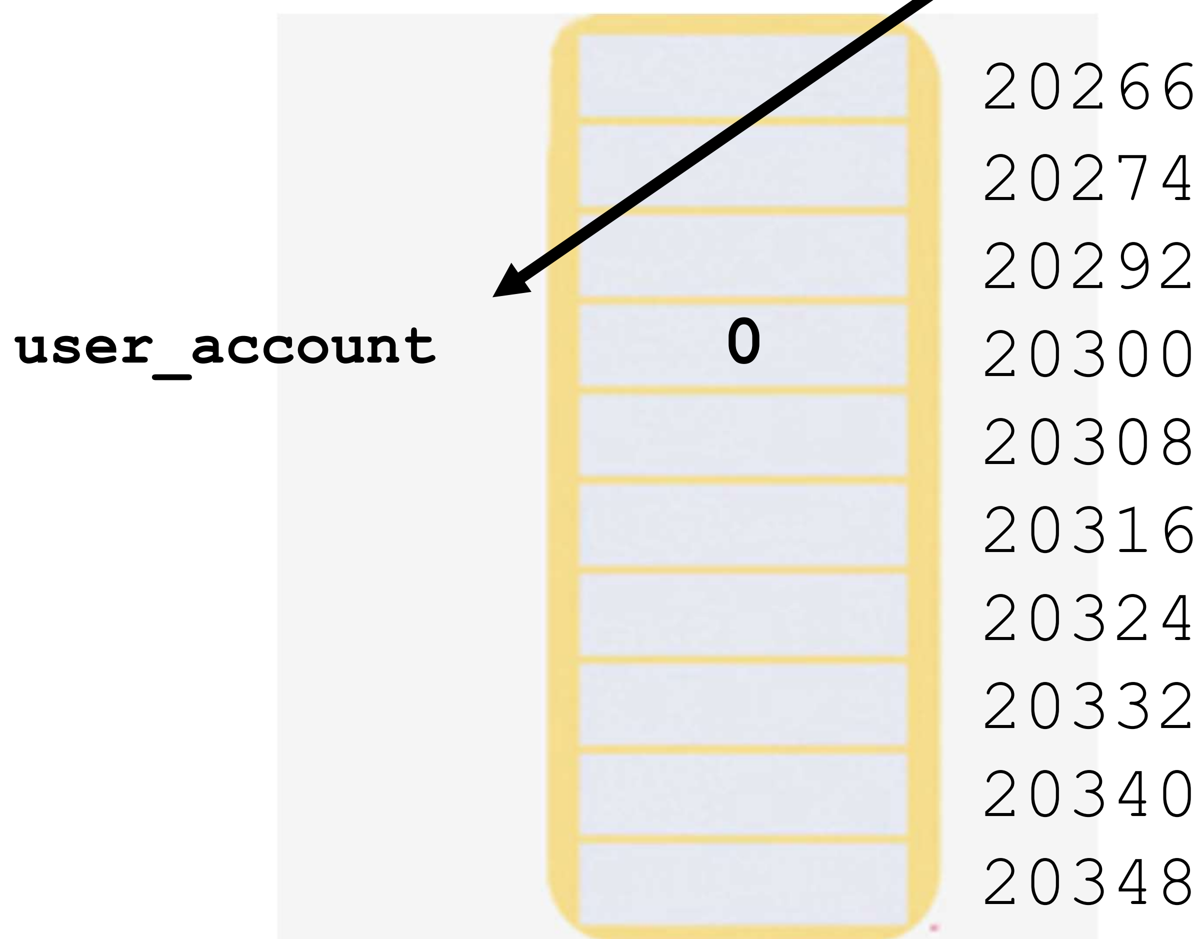
Addresses and Pointers

But really it's been like this all along:



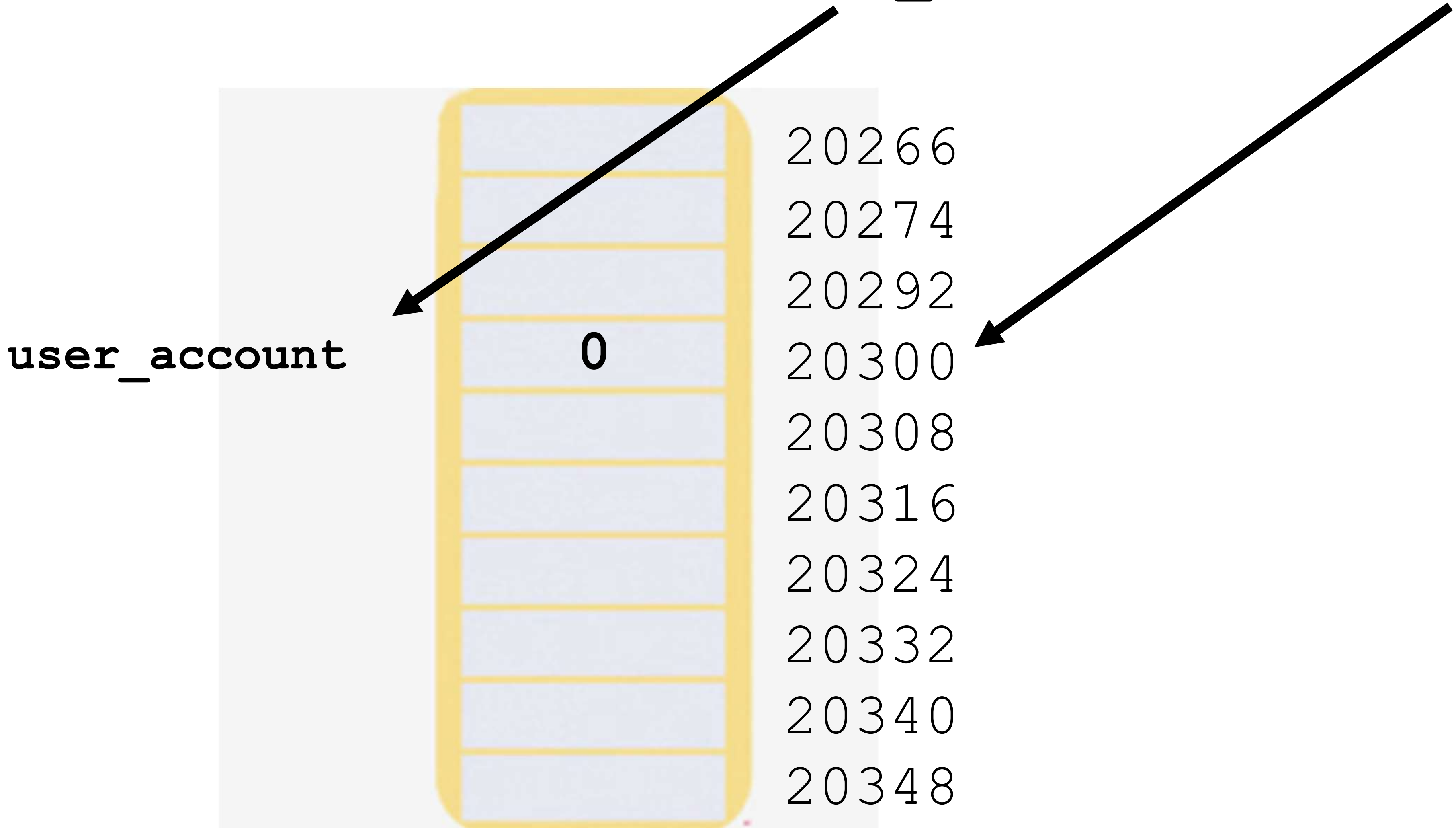
Addresses and Pointers

The address of the variable named user**_account**



Addresses and Pointers

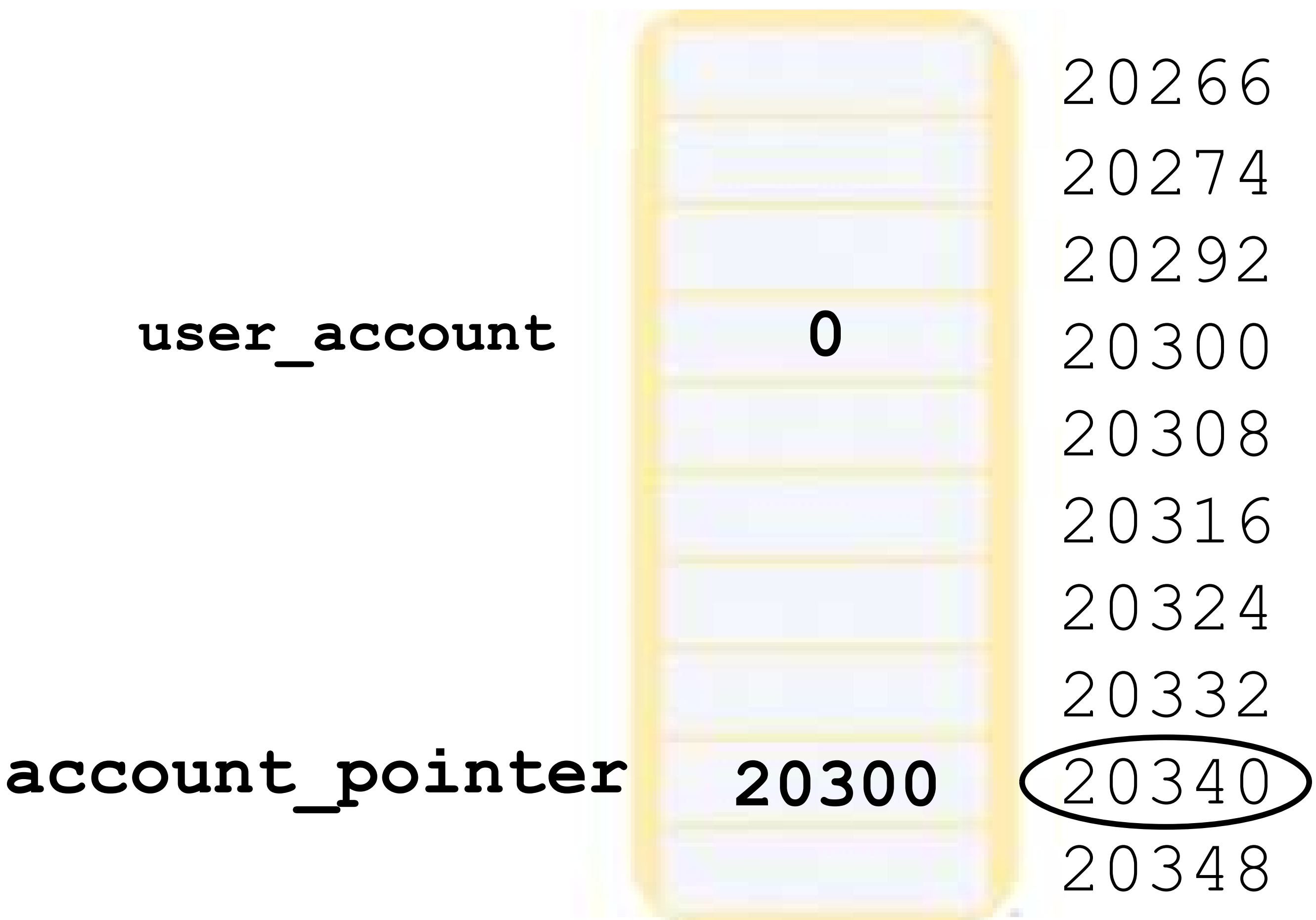
The address of the variable named **`user_account`** is **`20300`**



Addresses and Pointers

And, of course, **`account_pointer`** is *somewhere* in RAM:

double *account_pointer = &user_account;



Pointer Variables

- Definition:

  ```
  int  *intptr;
  ```

- Read as:
  "**`intptr`** can hold the address of an int" or "the variable that **`intptr`** points to has type int"

- The spacing in the definition does not matter:

  ```
  int * intptr;
  int*  intptr;
  ```

- **`*`** is called the indirection operator

Pointer Declaration

- Pointers are declared as follows:

```
<type> * variable_name ;
```

- e.g.

```
int  * xPtr;
```
// xPtr is a pointer to data of type integer

```
char * cPtr;
```
//cPtr is a pointer to data of type character

```
void * yPtr;
```
// yPtr is a generic pointer,
// represents any type

Pointer Assignment

- Assignment can be applied on pointers of the same type
- If not the same type, a cast operator must be used
- Exception: pointer to **void** does not need casting to convert a pointer to **void** type
- **void** pointers cannot be dereferenced
- Example

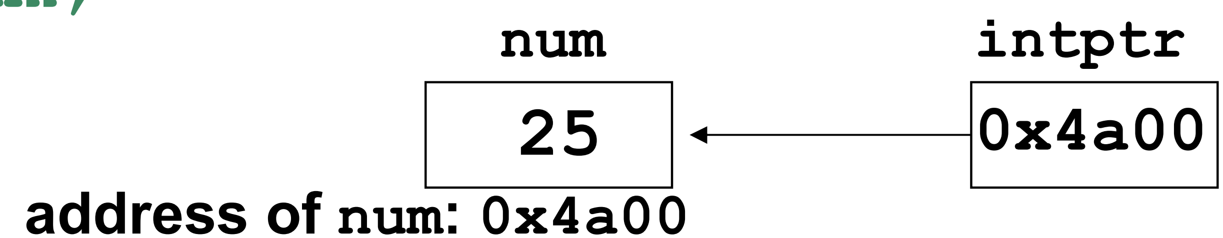
```
int *xPtr, *yPtr;
int x = 5;
xPtr = & x;      // xPtr now points to address of  x

yPtr = xPtr;    // now yPtr and xPtr point to x
```

Pointer Variables

- Definition and assignment:

```
int num = 25;
int *intptr;
intptr = &num;
```

- Memory layout:



- You can access **num** using **intptr** and indirection operator *****:

```
cout << intptr;   // prints 0x4a00
cout << *intptr; // prints 25
*intptr = 20;    // puts 20 in num
```

Program 9-2

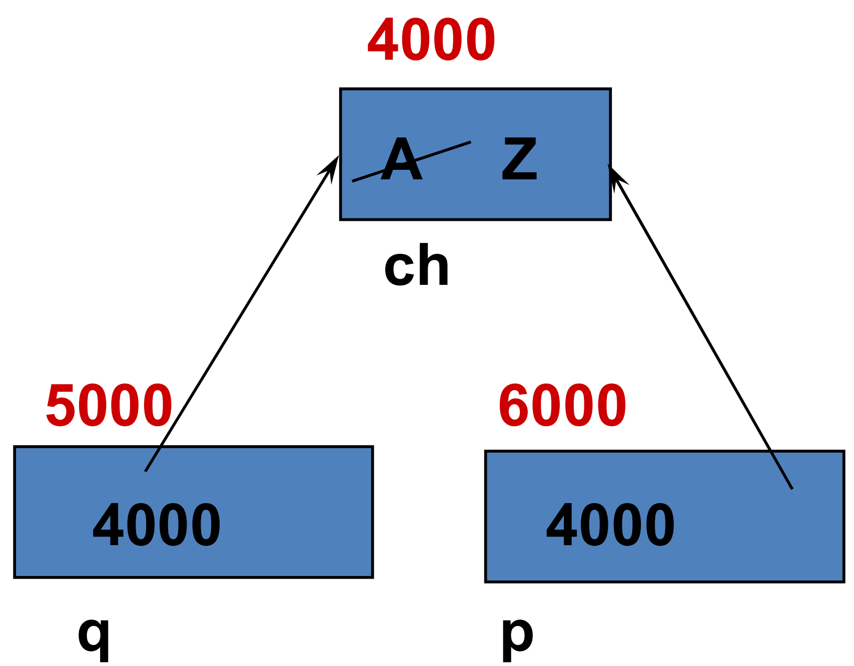
```
// This program stores the address of a variable in a pointer.
#include <iostream>
using namespace std;

int main()
{
   int x = 25;     // int variable
   int *ptr;       // Pointer variable, can point to an int

   ptr = &x;       // Store the address of x in ptr
   cout << "The value in x is " << x << endl;
   cout << "The address of x is " << ptr << endl;
   return 0;
}
```

Program Output

```
The value in x is 25
The address of x is 0x7e00
```

Another Example

```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;    // the right side has value 4000
          // now p and q both point to ch
```



Initializing Pointers

- You can initialize to NULL or 0 (zero)

```
int *ptr = NULL;
```

- You can initialize to addresses of other variables

```
int num, *numPtr = &num;
int Array[ASIZE], *valptr = Array;
```

- The initial value must have the correct type

```
float cost;
int *ptr = &cost; // won't work
```

The Indirection Operator

- The indirection operator (`*`) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25.

Program 9-3

```
// This program demonstrates the use of the indirection operator.
#include <iostream>
using namespace std;

int main()
{
   int x = 25;     // int variable
   int *ptr;       // Pointer variable, can point to an int

   ptr = &x;       // Store the address of x in ptr

   // Use both x and ptr to display the value in x.
   cout << "Here is the value in x, printed twice:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x

   // Assign 100 to the location pointed to by ptr. This
   // will actually assign 100 to x.
   *ptr = 100;

   // Use both x and ptr to display the value in x.
   cout << "Once again, here is the value in x:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x
   return 0;
}
```

The `NULL` Pointer

There is a pointer constant called the "null pointer" denoted by NULL in cstddef.

But NULL is not memory address 0.

NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer's job to check for this.

```
while (ptr != NULL)
{
   . . .          // ok to use *ptr here
}
```

Comparing Pointers

- Relational operators can be used to compare the addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                      // addresses
if (*ptr1 == *ptr2) // compares
                    // contents
```

Something Like Pointers: Arrays

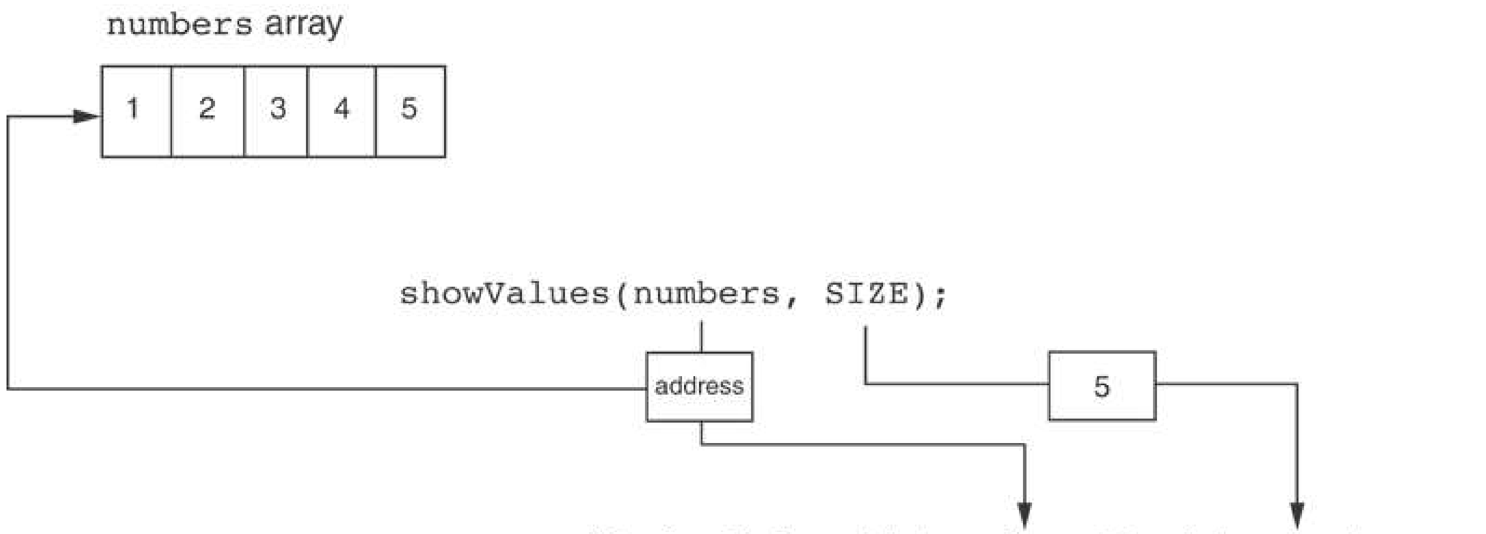
- We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.

- For example, suppose we use this statement to pass the array `numbers` to the `showValues` function:

```
showValues(numbers, SIZE);
```

Something Like Pointers : Arrays

The `values` parameter, in the `showValues` function, points to the `numbers` array.




```
void showValues(int values[], int size)
{
    for (int count = 0; count < size; count++)
        cout << values[count] << endl;
}
```

C++ automatically stores the address of `numbers` in the `values` parameter.

Something Like Pointers: Reference Variables

- We have also worked with something like pointers when we learned to use reference variables. Suppose we have this function:

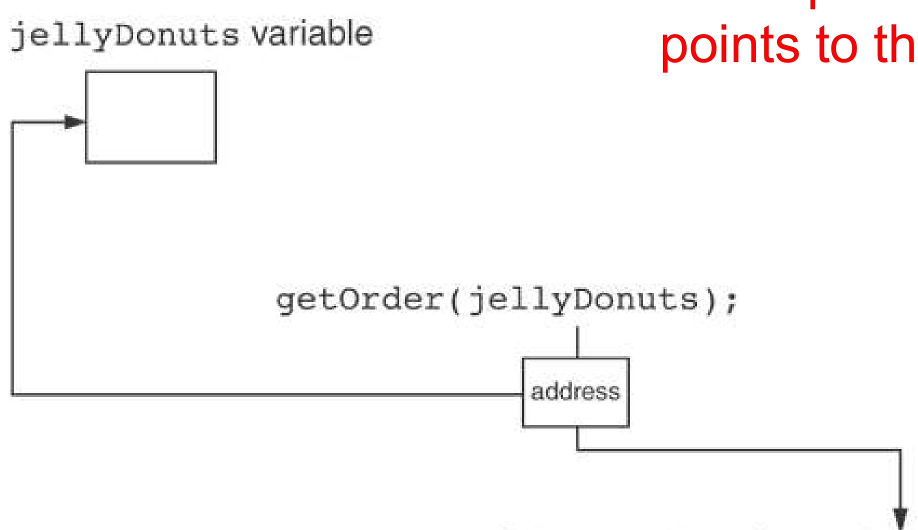
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

- And we call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts);
```

Something Like Pointers: Reference Variables

The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.




```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```

4	7	11

starting address of `vals`: `0x4a00`

```
cout << vals;        // displays
                     // 0x4a00
cout << vals[0];     // displays 4
```

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Program 9-5

```
// This program shows an array name being dereferenced with the *
// operator.
#include <iostream>
using namespace std;

int main()
{
   short numbers[] = {10, 20, 30, 40, 50};

   cout << "The first element of the array is ";
   cout << *numbers << endl;
   return 0;
}
```

Program Output

```
The first element of the array is 10
```

Pointers in Expressions

Given:

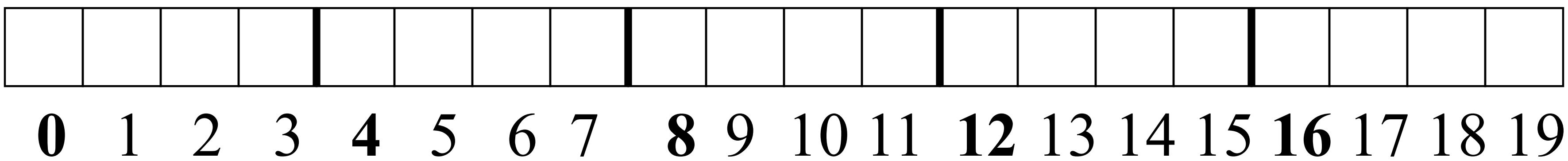
```
int vals[]={4,7,11}, *valptr;
valptr = vals;
```

What is `valptr + 1`? It means (address in `valptr`) + (1 * size of an int)

```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use ( ) as shown in the expressions

A picture of int arr[5]



- Assuming **arr==0** (it holds the address 0):
  - **arr[0]** is at address 0 – which is equal to **arr**
  - **arr[1]** is at address 4 – which is equal to **arr+1**
  - **arr[2]** is at address 8– which is equal to **arr+2**
  - **arr[3]** is at address 12– which is equal to **arr+3**
  - **arr[4]** is at address 16– which is equal to **arr+4**

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

Array Access

- Conversion: `vals[i]` is equivalent to `*(vals + i)`
- No bounds checking performed on array access, whether using array name or a pointer

From Program 9-7

```
const int NUM_COINS = 5;
double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
double *doublePtr;    // Pointer to a double
int count;            // Array index

// Assign the address of the coins array to doublePtr.
doublePtr = coins;

// Display the contents of the coins array. Use subscripts
// with the pointer!
cout << "Here are the values in the coins array:\n";
for (count = 0; count < NUM_COINS; count++)
   cout << doublePtr[count] << " ";

// Display the contents of the array again, but this time
// use pointer notation with the array name!
cout << "\nAnd here they are again:\n";
for (count = 0; count < NUM_COINS; count++)
   cout << *(coins + count) << " ";
cout << endl;
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
```

Pointer Arithmetic

- Increment / decrement pointers (++ or --)
- Add / subtract an integer to/from a pointer

  (+ or += , - or -=)
- Pointers may be subtracted from each other
- Pointer arithmetic is meaningless unless performed on an array

Pointer Arithmetic

- Operations on pointer variables:

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;   // points at 11`

`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

From Program 9-9

```
    const int SIZE = 8;
    int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *numPtr;    // Pointer
    int count;      // Counter variable for loops

    // Make numPtr point to the set array.
    numPtr = set;

    // Use the pointer to display the array contents.
    cout << "The numbers in set are:\n";
    for (count = 0; count < SIZE; count++)
    {
       cout << *numPtr << " ";
       numPtr++;
    }

    // Display the array contents in reverse order.
    cout << "\nThe numbers in set backward are:\n";
    for (count = 0; count < SIZE; count++)
    {
       numPtr--;
       cout << *numPtr << " ";
    }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); // ptr is pointer to an int
```

2) asterisk ***** in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function

```
getNum(&num);     // pass address of num to getNum
```

Example

```
void swap(int *x, int *y)
{  int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Examples on Pointers

```
//File: swap.cpp
//A program to call a function to swap two numbers using reference parameters

 #include <iostream.h>
void swap(int *, int *);   // This is swap's prototype
void  main()
{    int x = 5, y = 7;
     swap(&x , &y);         // calling swap with reference parameters
     cout << "\n x is now "<< x << " and y is now " << y << '\n';
}
// swap function is defined here using dereferencing operator '*'
void swap(int *a, int *b)
{    int temp;
     temp = *a;
     *a = *b;
     *b = temp;
}
```

Program 9-11

```
// This program uses two functions that accept addresses of
// variables as arguments.
#include <iostream>
using namespace std;

// Function prototypes
void getNumber(int *);
void doubleValue(int *);

int main()
{
   int number;

   // Call getNumber and pass the address of number.
   getNumber(&number);

   // Call doubleValue and pass the address of number.
   doubleValue(&number);

   // Display the value in number.
   cout << "That value doubled is " << number << endl;
   return 0;
}

```

(Program Continues)

Program 9-11 *(continued)*

```
//*****************************************************************
// Definition of getNumber. The parameter, input, is a pointer. *
// This function asks the user for a number. The value entered  *
// is stored in the variable pointed to by input.               *
//*****************************************************************

void getNumber(int *input)
{
   cout << "Enter an integer number: ";
   cin >> *input;
}

//*****************************************************************
// Definition of doubleValue. The parameter, val, is a pointer. *
// This function multiplies the variable pointed to by val by   *
// two.                                                         *
//*****************************************************************

void doubleValue(int *val)
{
   *val *= 2;
}
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 10 [Enter]
That value doubled is 20
```

Examples on Pointers (Cont.)

```
#include <iostream.h>
void Increment(int*);

void main() {
  int A = 10;
  Increment(&A);
   cout<<A<<endl;
 }

void Increment(int *X) { ++*X; }
```

When calling, the pointer formal parameter will points to the actual parameter.

Examples on Pointers (Cont.)

```
//File: pointers.cpp
//A program to test pointers and references
#include <iostream.h>
void main ( )
{   int intVar = 10;
    int  *intPtr;    // intPtr is a pointer
    intPtr = & intVar;
    cout << "\nLocation of intVar: " << & intVar;
    cout << "\nContents of intVar: " << intVar;
    cout << "\nLocation of intPtr: " << & intPtr;
    cout << "\nContents of intPtr: " << intPtr;
    cout << "\nThe value that intPtr points to: " << * intPtr;
}
```

Pointers to Constants

- Example: Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] =
     { 18.55, 17.45, 12.85,
       14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an array of constant doubles.

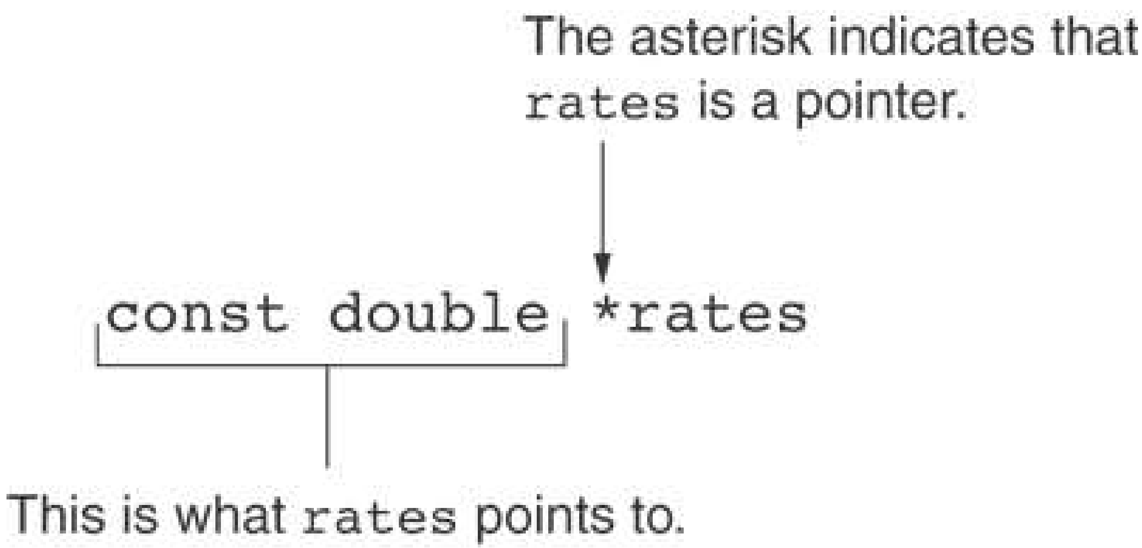
Pointers to Constants

- Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```
void displayPayRates(const double *rates, int size)
{
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

The parameter, rates, is a pointer to `const double`.

Declaration of a Pointer to Constant



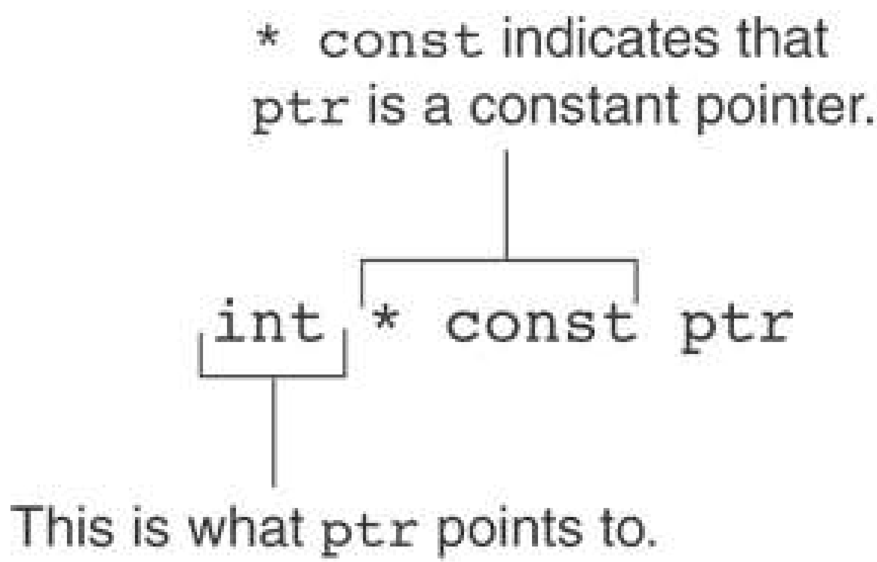
Constant Pointers

- A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

- Example

```
int value = 22;
int * const ptr = &value;
```

Constant Pointers



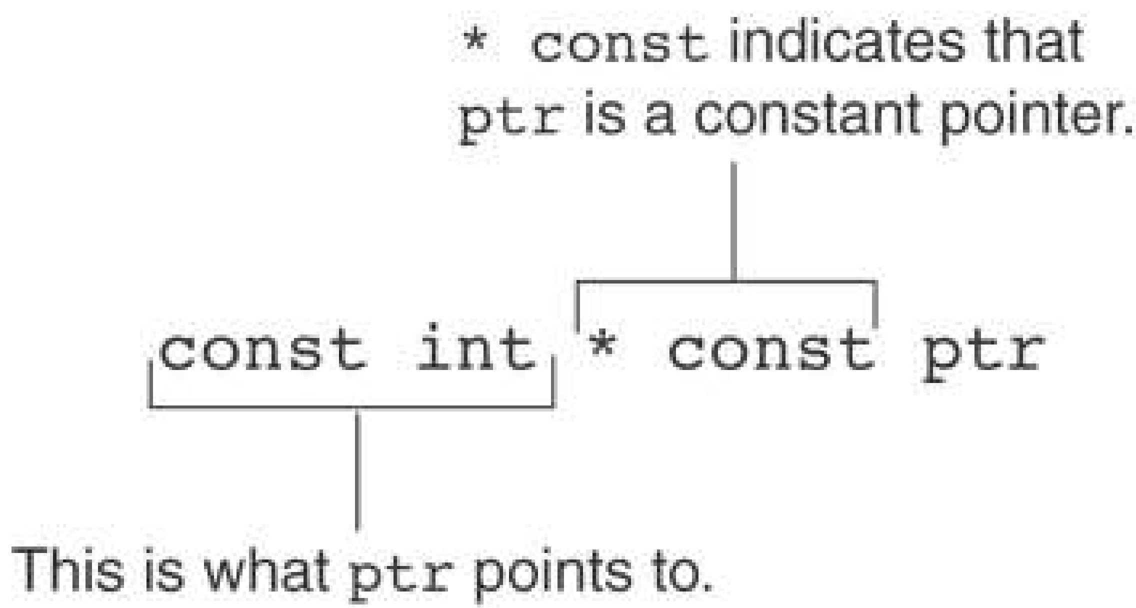
Constant Pointers to Constants

- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

```
int value = 22;
const int * const ptr = &value;
```

Constant Pointers to Constants



Allocation of memory

STATIC ALLOCATION	DYNAMIC ALLOCATION
Static allocation is the allocation of memory space at compile time.	**Dynamic allocation is the allocation of memory space at run time by using operator `new`.**

Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses `new` operator to allocate memory:

```
double *dptr;
dptr = new double;
```

- `new` returns address of memory location

Dynamic Memory Allocation

- Can also use `new` to allocate array:
  ```
  const int SIZE = 25;
  arrayPtr = new double[SIZE];
  ```
- Can then use `[]` or pointer arithmetic to access array:
  ```
  for(i = 0; i < SIZE; i++)
      *arrayptr[i] = i * i;
  ```
  or
  ```
  for(i = 0; i < SIZE; i++)
      *(arrayptr + i) = i * i;
  ```
- Program will terminate if not enough memory available to allocate

Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Program 9-14

```
// This program totals and averages the sales figures for any
// number of days. The figures are stored in a dynamically
// allocated array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   double *sales,         // To dynamically allocate an array
          total = 0.0,    // Accumulator
          average;        // To hold average sales
```

Program 9-14 *(continued)*

```
    int numDays,           // To hold the number of days of sales
        count;             // Counter variable

    // Get the number of days of sales.
    cout << "How many days of sales figures do you wish ";
    cout << "to process? ";
    cin >> numDays;

    // Dynamically allocate an array large enough to hold
    // that many days of sales amounts.
    sales = new double[numDays];

    // Get the sales figures for each day.
    cout << "Enter the sales figures below.\n";
    for (count = 0; count < numDays; count++)
    {
       cout << "Day " << (count + 1) << ": ";
       cin >> sales[count];
    }

```

Program 9-14 (Continued)

```
    // Calculate the total sales
    for (count = 0; count < numDays; count++)
    {
       total += sales[count];
    }

    // Calculate the average sales per day
    average = total / numDays;

    // Display the results
    cout << fixed << showpoint << setprecision(2);
    cout << "\n\nTotal Sales: $" << total << endl;
    cout << "Average Sales: $" << average << endl;

    // Free dynamically allocated memory
    delete [] sales;
    sales = 0;         // Make sales point to null.

    return 0;
}
```

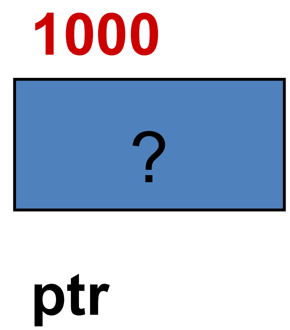
Program Output with Example Input Shown in Bold

```
How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

Total Sales: $4067.13
Average Sales: $813.43
```

Notice that in line 49 the value 0 is assigned to the `sales` pointer. It is a good practice to store 0 in a pointer variable after using delete on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents errors from occurring if `delete` is accidentally called on the pointer again. The `delete` operator is designed to have no effect when used on a null pointer.

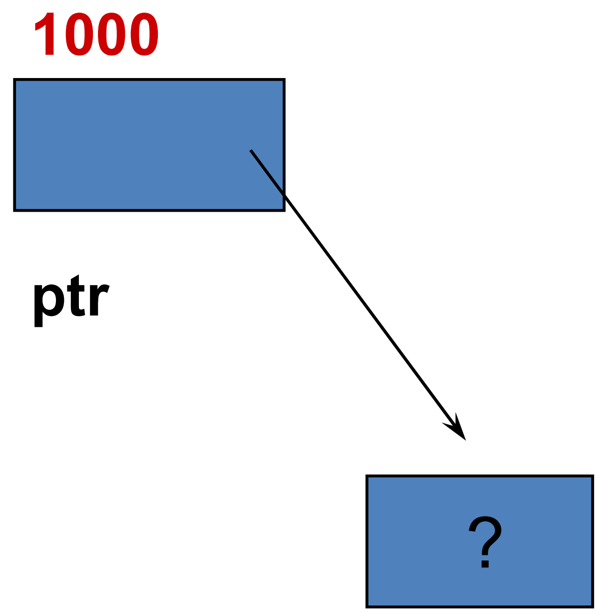
Dynamically Allocated Data

```
char*  ptr;

ptr = new char;
*ptr = 'B';
cout << *ptr;
delete  ptr;
```



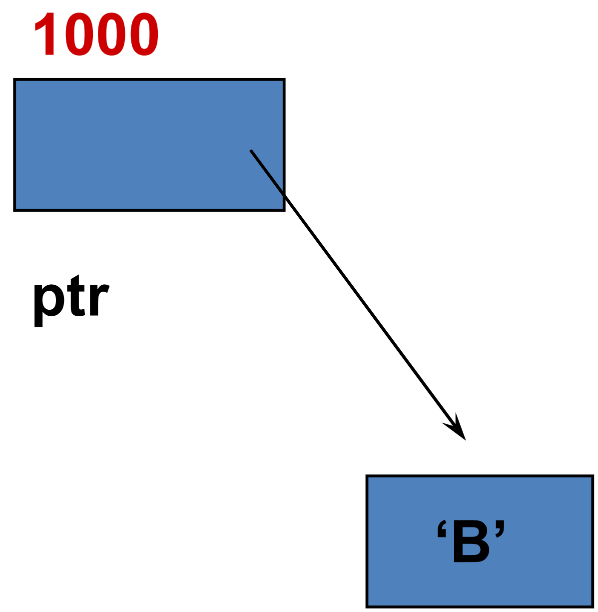
Dynamically Allocated Data

```
char*  ptr;

ptr = new char;
*ptr = 'B';

cout << *ptr;
delete  ptr;
```



Dynamically Allocated Data

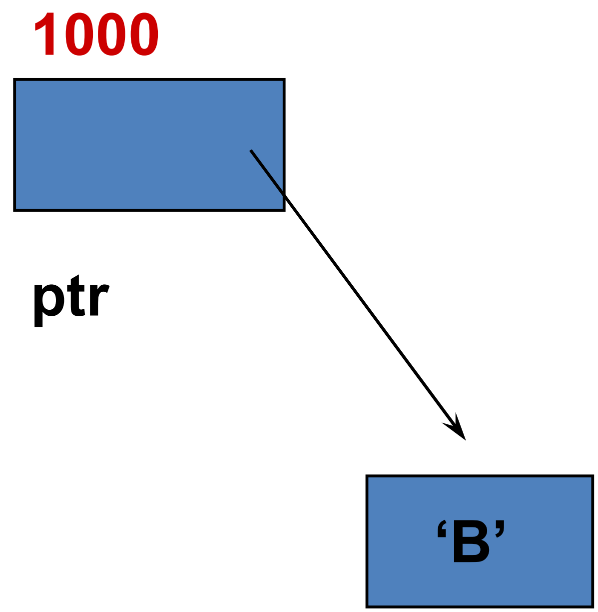
```
char*  ptr;

ptr = new char;

*ptr = 'B';

cout << *ptr;
delete  ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;


ptr = new char;

*ptr = 'B';

cout << *ptr;
delete  ptr;
```

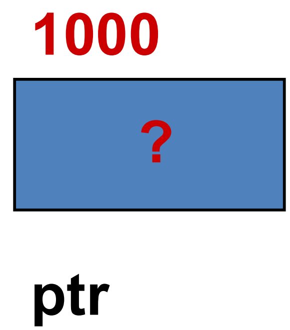


Displays B

NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
char*  ptr;

ptr = new char;

*ptr = ‘B’;

cout << *ptr;

delete  ptr;
```



NOTE: Delete deallocates the memory pointed to by ptr.

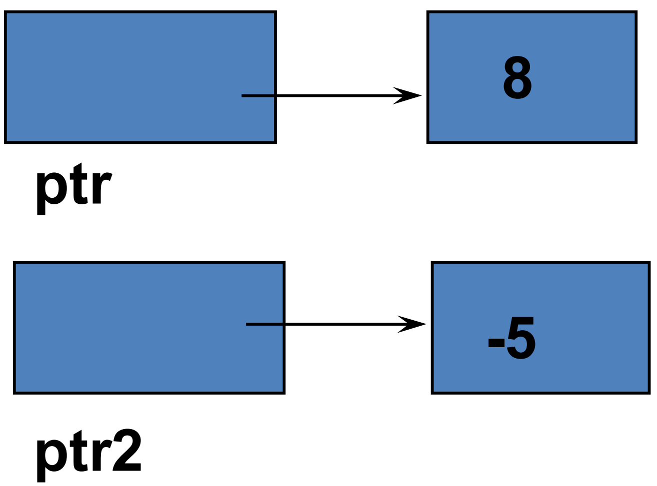
Dangling Pointers and Memory Leaks

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Solution: set such pointers to NULL (or **`nullptr`** in C++ 11) as soon as the memory is freed.
- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use

Memory Leak

A memory leak occurs when dynamic memory (that was created using operator `new`) has been left without a pointer to it by the programmer, and so is inaccessible.

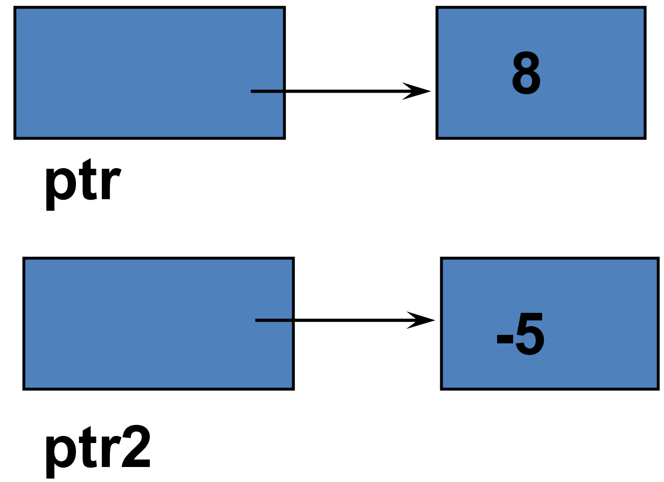
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



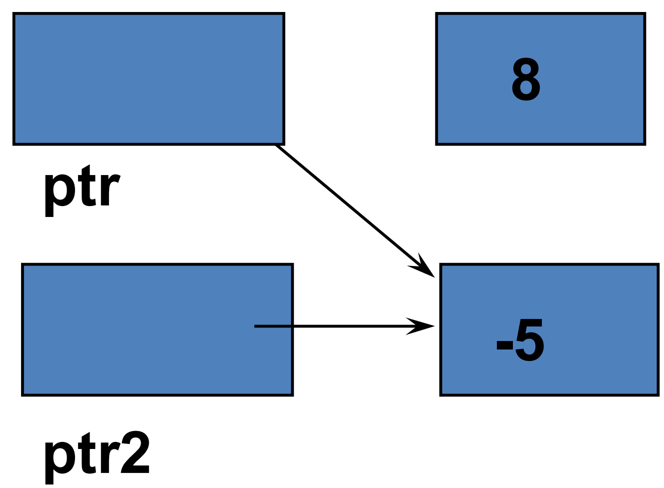
How else can an object become inaccessible?

Causing a Memory Leak

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



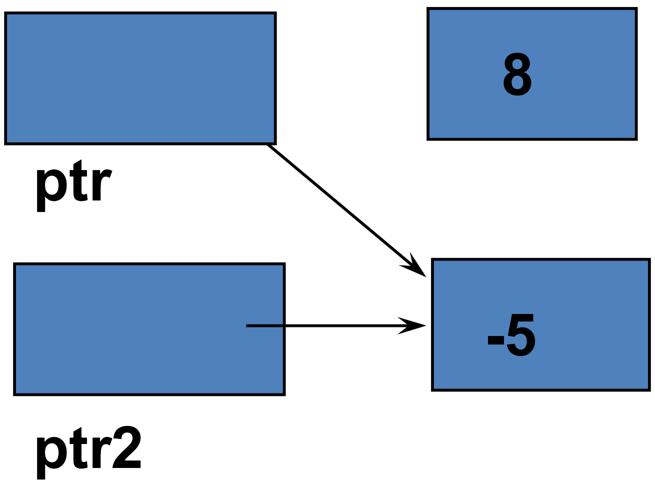

```
ptr = ptr2;
```
// here the 8 becomes inaccessible



A Dangling Pointer

occurs when two pointers point to the same object and delete is applied to one of them.

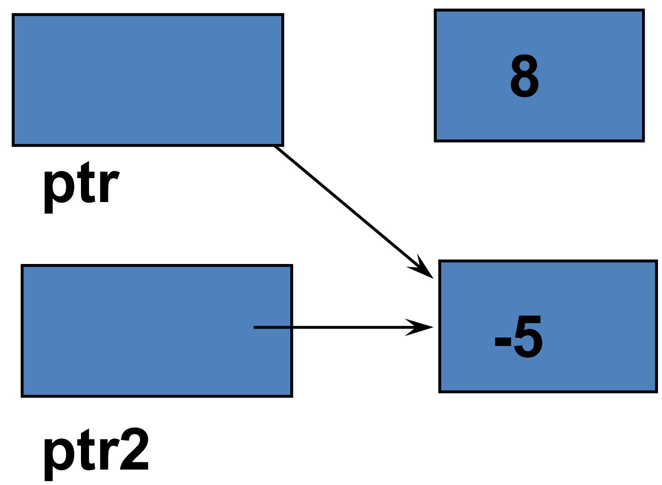
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```

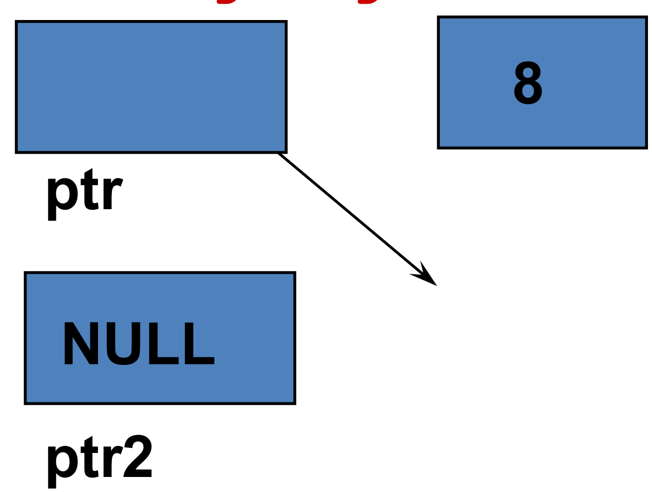


FOR EXAMPLE,

Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;

delete ptr2;      // ptr is left dangling
ptr2 = NULL;
```





More on Memory Leaks

General guidelines to avoid memory leaks:

- If a function allocates memory via **`new`**, it should, whenever possible, also deallocate the memory using **`delete`**
- If a class needs dynamic memory, it should
  - allocate it using **`new`** in the constructor
  - deallocate it using **`delete`** in the destructor

Returning Pointers from Functions

- Pointer can be the return type of a function:

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function.
- A function should only return a pointer:
  - to data that was passed to the function as an argument, or
  - to dynamically allocated memory

From Program 9-15

```
int *getRandomNumbers(int num)
{
   int *array;     // Array to hold the numbers

   // Return null if num is zero or negative.
   if (num <= 0)
      return NULL;

   // Dynamically allocate the array.
   array = new int[num];

   // Seed the random number generator by passing
   // the return value of time(0) to srand.
   srand( time(0) );

   // Populate the array with random numbers.
   for (int count = 0; count < num; count++)
      array[count] = rand();

   // Return a pointer to the array.
   return array;
}
```

Thank you

National University
Of Computer and Emerging Sciences