Pointers

CS 110C
Anita Rathi

Topics

- Pointers and the Address Operator
- Pointer Variables
- The Relationship Between Arrays and Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers

Topics (continued)

Pointers as Function Parameters

Pointers to Constants and Constant Pointers

Dynamic Memory Allocation

Returning Pointers from Functions

Pointers to Class Objects and Structures

Selecting Members of Objects

Pointers and the Address Operator

- Each variable in a program is stored at a unique location in memory that has an address.
- Use the address operator **&** to get the address of a variable:

```
int num = -23;
cout << &num; // prints address
              // in hexadecimal
```

- The address of a memory location is a pointer

Pointer Variables

- Pointer variable (pointer): a variable that holds an address.
- Pointers provide an alternate way to access memory locations.

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 Variables

- Assignment:

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

- Memory layout:

address of **`num`**: `0x4a00`

- Can access **`num`** using **`intptr`** and indirection operator **`*`**:

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

The Relationship Between Arrays and Pointers

An array name is the starting address of the 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

- An array name can be used as a pointer constant.

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

- A pointer can be used as an array name.

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

Pointers in Expressions

- Given:

```
int vals[]={4,7,11};
int *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 **`( )`** in expression

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

- Array notation

  `vals[i]`

  is equivalent to the pointer notation

  `*(vals + i)`

- No bounds checking is performed on array access

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

– Increment and decrement operators **++**, **− −**

– Integers can be added to or subtracted from pointers using the operators **+**, **−**, **+=**, and **−=**

– One pointer can be subtracted from another by using the subtraction operator **−**

Pointer Arithmetic

Assume the variable definitions

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

Examples of use of **++** and **--**

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

More on Pointer Arithmetic

Assume the variable definitions:

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

Example of the use of **+** to add an int to a pointer:

```
cout << *(valptr + 2)
```

This statement will print 11

More on Pointer Arithmetic

Assume the variable definitions:

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

Example of use of +=:

```
valptr = vals; // points at 4
valptr += 2;   // points at 11
```

More on Pointer Arithmetic

Assume the variable definitions

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

Example of pointer subtraction

```
valptr += 2;
cout << valptr - val;
```

This statement prints **`2`**: the number of **`int`**s between **`valptr`** and **`val`**

Initializing Pointers

- Can initialize to NULL or 0 (zero)

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

- Can initialize to addresses of other variables

```
int num, *numPtr = &num;
int val[ISIZE], *valptr = val;
```

- Initial value must have correct type

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

Comparing Pointers

- Relational operators can be used to compare 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
```

Pointers as Function Parameters

- A pointer can be a parameter
- It works like a reference parameter to allow changes to argument from within a function
- A pointer parameter must be explicitly dereferenced to access the contents at that address

Pointers as Function Parameters

Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr);
```

2) asterisk * in body to dereference the pointer

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

3) address as argument to the function in the call

```
getNum(&num);
```

Pointers as Function Parameters

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int num1 = 2, num2 = -3;
swap(&num1, &num2);  //call
```

Ponters to Constants and Constant Pointers

- Pointer to a constant: cannot change the value that is pointed at.

- Constant pointer: the address in the pointer cannot change after the pointer is initialized.

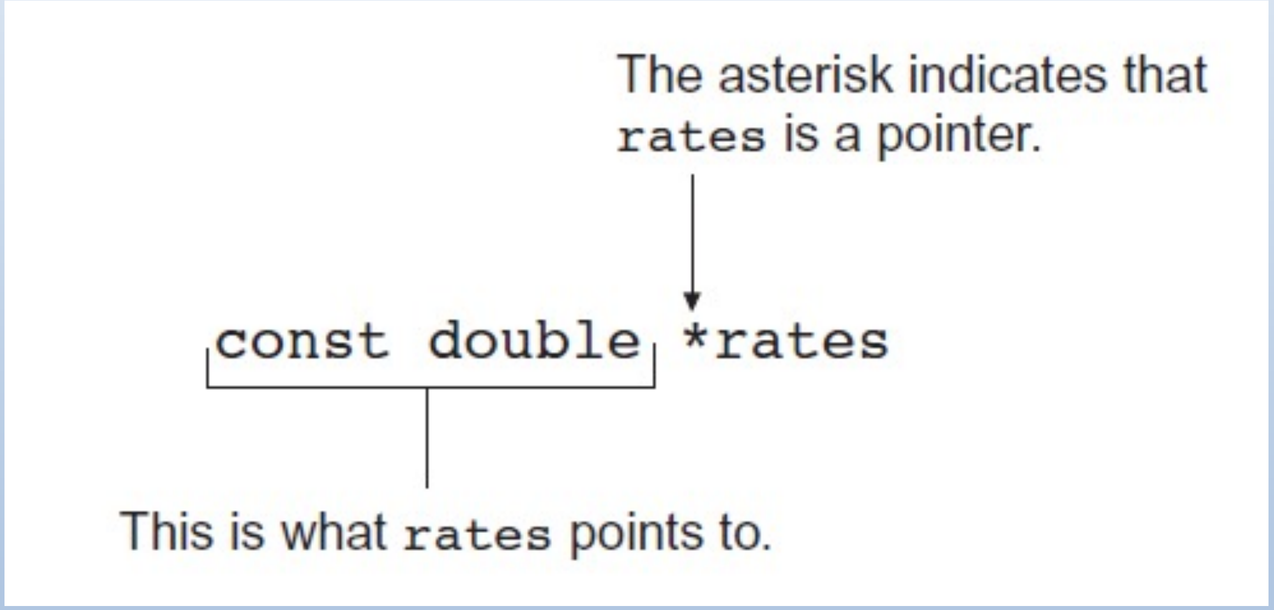
Ponters to Constant

- Must use **`const`** keyword in pointer definition:

```
const double taxRates[] =
                    {0.65, 0.8, 0.75};
const double *ratePtr;
```

- Use **`const`** keyword for pointers in function headers to protect data from modification from within function.

Pointer to Constant – What does the Definition Mean?



Read as: “rates is a pointer to a constant that is a double.”

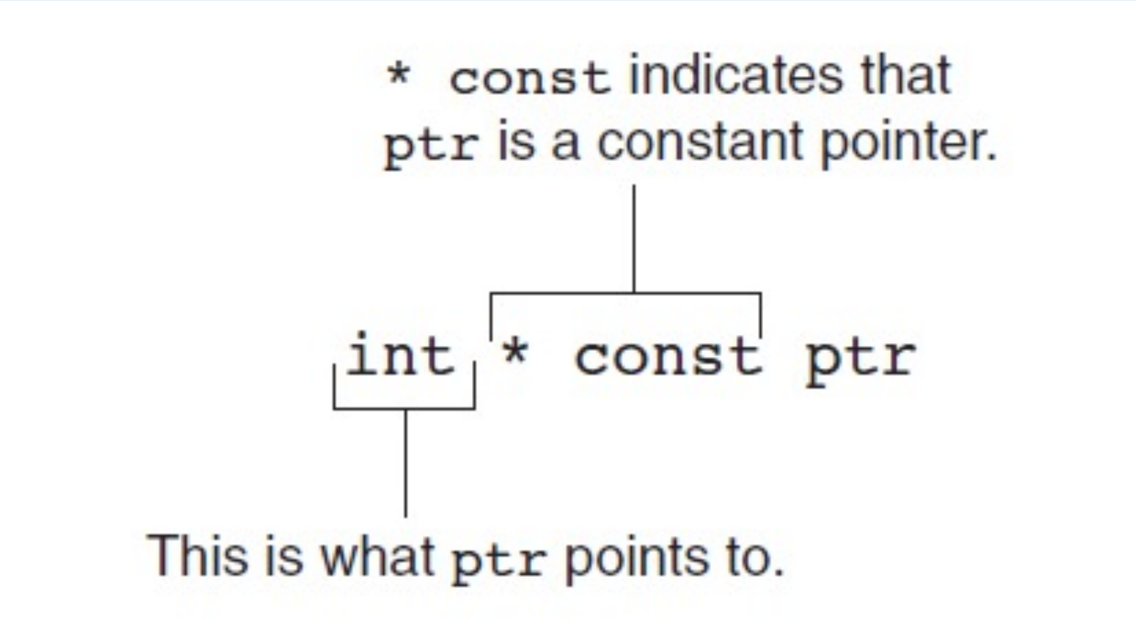
Constant Pointers

- Defined with **`const`** keyword adjacent to variable name:

```
int classSize = 24;
int * const classPtr = &classSize;
```

- Must be initialized when defined
- Can be used without initialization as a function parameter
  - Initialized by argument when function is called
  - Function can receive different arguments on different calls
- While the <u>address</u> in the pointer cannot change, the <u>data</u> at that address may be changed

Constant Pointer – What does the Definition Mean?



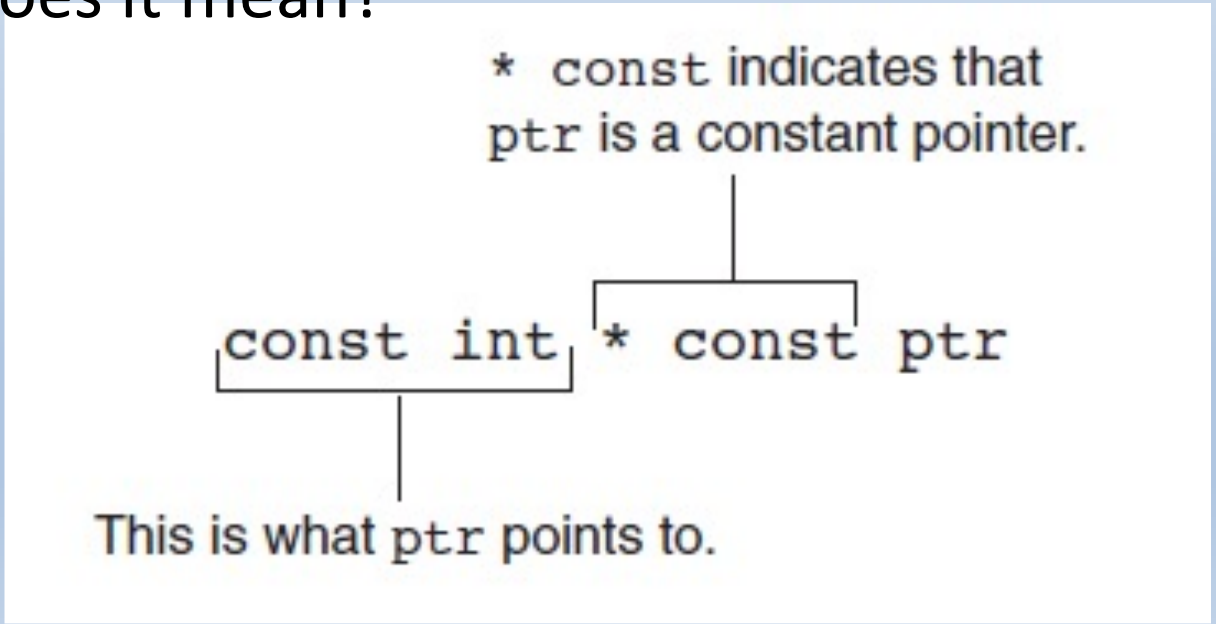
Read as: “pts is a constant pointer to an int.”

Constant Pointer to Constant

- Can combine pointer to constants and constant pointers:

```
int size = 10;
const int * const ptr = &size;
```

- What does it mean?



Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- 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

  ```
  arrayPtr = new double[25];
  ```

  – Program may terminate if there is not sufficient memory

- Can then use **`[ ]`** or pointer arithmetic to access array

Dynamic Memory Example

```
int *count, *arrayptr;
count = new int;
cout <<"How many students? ";
cin >> *count;
arrayptr = new int[*count];

for (int i=0; i<*count; i++)
{
  cout << "Enter score " << i << ": ";
  cin >> arrayptr[i];
}
```

Releasing Dynamic Memory

- Use **`delete`** to free dynamic memory

  ```
  delete count;
  ```

- Use **`delete []`** to free dynamic array memory

  ```
  delete [] arrayptr;
  ```

- Only use **`delete`** with dynamic memory!

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 0 as soon as 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

Returning Pointers from Functions

- Pointer can be return type of function

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

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

Pointers to Class Objects and Structures

- Can create pointers to objects and structure variables

```
struct Student {…};
class Square {…};
Student stu1;
Student *stuPtr = &stu1;
Square sq1[4];
Square *squarePtr = &sq1[0];
```

- Need to use `()` when using `*` and `.` operators

```
(*stuPtr).studentID = 12204;
```

Structure Pointer Operator

- Simpler notation than **`(*ptr).member`**
- Use the form **`ptr->member`**:

```
stuPtr->studentID = 12204;
squarePtr->setSide(14);
```

in place of the form **`(*ptr).member`**:

```
(*stuPtr).studentID = 12204;
(*squarePtr).setSide(14);
```

Dynamic Memory with Objects

- Can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- Can pass values to constructor:

```
squarePtr = new Square(17);
```

- **`delete`** causes destructor to be invoked:

```
delete squarePtr;
```

Structure/Object Pointers as Function Parameters

- Pointers to structures or objects can be passed as parameters to functions
- Such pointers provide a pass-by-reference parameter mechanism
- Pointers must be dereferenced in the function to access the member fields

Controlling Memory Leaks

- Memory that is allocated with **`new`** should be deallocated with a call to **`delete`** as soon as the memory is no longer needed. This is best done in the same function as the one that allocated the memory.
- For dynamically-created objects, **`new`** should be used in the constructor and **`delete`** should be used in the destructor

Selecting Members of Objects

Situation: A structure/object contains a pointer as a member. There is also a pointer to the structure/object.

Problem: How do we access the pointer member via the structure/object pointer?

```
struct GradeList
  { string courseNum;
    int * grades;
  }
GradeList test1, *testPtr = &test1;
```

Selecting Members of Objects

Expression	Meaning
`testPtr->grades`	Access the grades pointer in **`test1`**. This is the same as **`(*testPtr).grades`**
`*testPtr->grades`	Access the value pointed at by **`testPtr->grades`**. This is the same as **`*(*testPtr).grades`**
`*test1.grades`	Access the value pointed at by **`test1.grades`**

Thank you