Pointers (Chapter 9 & 11.9)

Dr. Yingwu Zhu

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

Getting the Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator & to get address of a variable:

```
int num = -99;
cout << &num; // prints address
              // in hexadecimal
```

Pointer Variables

- Pointer variable : Often just called a pointer, it's a variable that holds an address
  - Its content is a memory address
- Because a pointer variable holds the address of another piece of data, it "points" to the data
  - Analogy: an entry in a yellow book

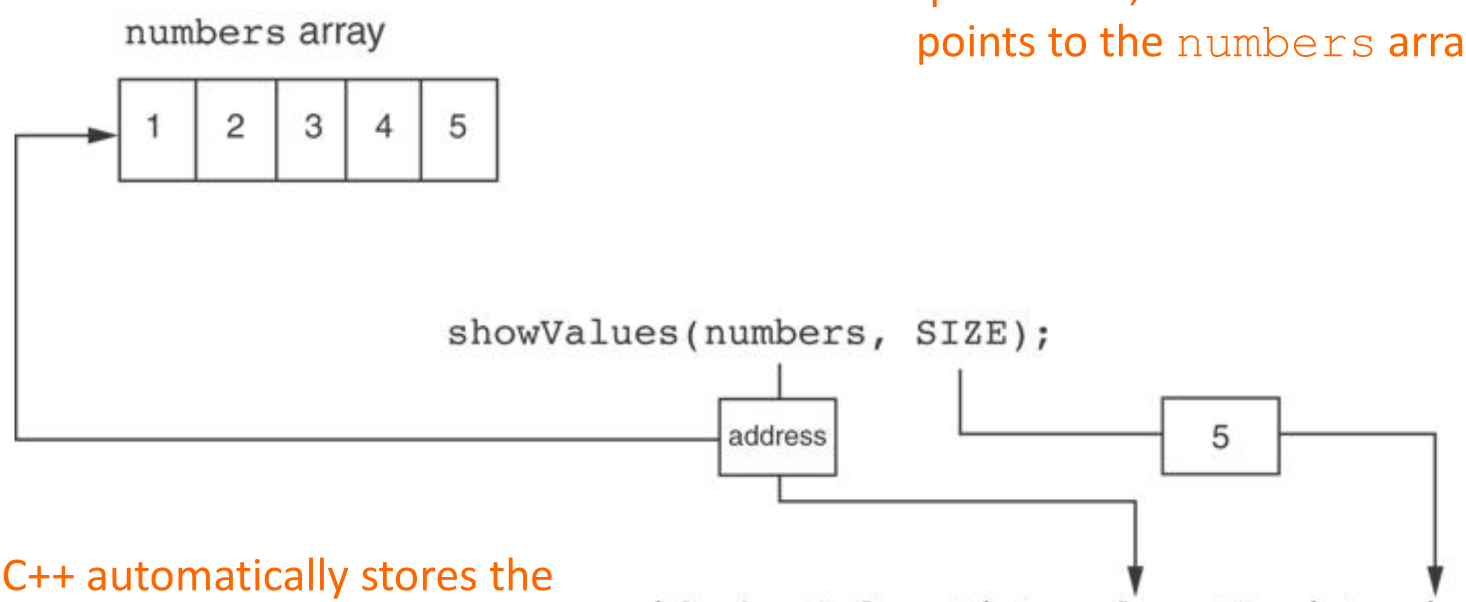
Something Like Pointers: Arrays

- 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.



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

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

```
void showValues(int* values, int size) {
    .....
}
```

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:

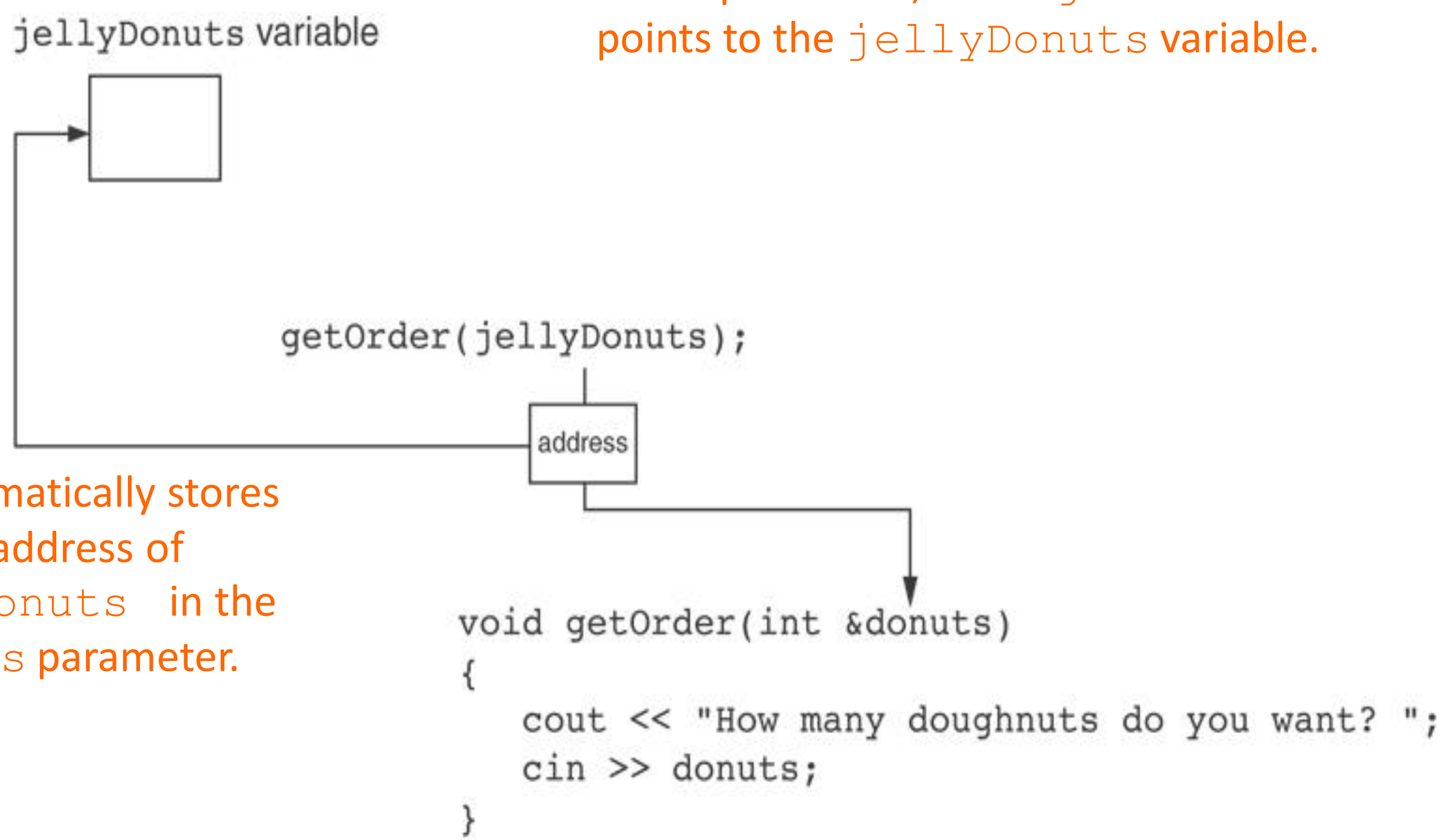
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

- 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.



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

Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.

Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:

  "`intptr` can hold the address of an int"

- Spacing in definition does not matter:

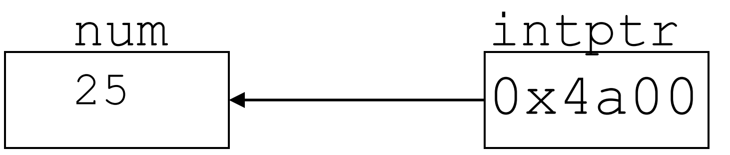
```
int * intptr;  // same as above
int*  intptr;  // same as above
```

Pointer Variables

- Assigning an address to a pointer variable:

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

- Memory layout:



address of `num`: `0x4a00`

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
```

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;
}
```

Program 9-3 *(continued)*

Program Output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```

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
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

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
```

Pointers & Arrays

```
int array[] = {1,2,3,4};
int *ptr = NULL;
ptr = array;
ptr = &array[0]; /*same as line above*/
printf("array[0]  = %d\n",array[0]);
printf("*(array+1)= %d\n",*(array+1));
printf("ptr[2]    = %d\n",ptr[2]);
printf("*(ptr+3)  = %d\n",*(ptr+3));
```

What is the output?

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 * sizeof(int))

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

Must use ( ) as shown in the expressions

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)`
- Error prone: No bounds checking performed on array access, whether using array name or a pointer

```
   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
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

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`

```
    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 << " ";
    }
```

```
    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
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

Initializing Pointers

- Can initialize at definition time:
```
int num, *numptr = &num;
int val[3], *valptr = val;
```
- Cannot mix data types:
```
double cost;
int *ptr = &cost; // won't work
```
- Can test for an invalid address for `ptr` with:
```
if (!ptr) ...
```
- Take-home message:
  - initialize a pointer to be NULL
  - Check validity of a pointer before using it

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

Comparing Pointers

- Relational operators (<, >=, etc.) 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
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

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);
```

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

Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.

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.

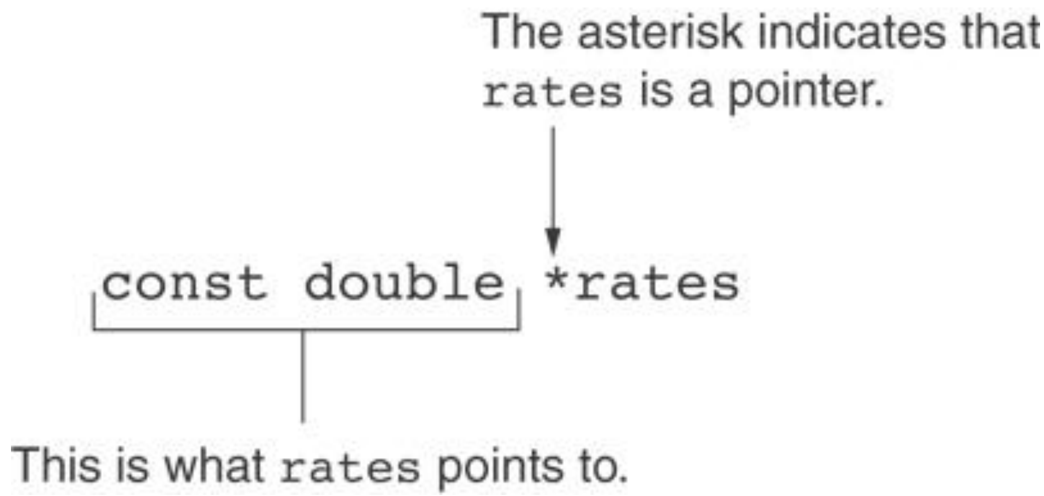
Pointers to Constants

- 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



Pointers to Constant in Functions

```
void swap2(const int *a, const int *b){
/*cannot change value at that address*/

  int temp = *a; /*no warning*/
  *a=*b;         /*gives warning*/
  *b=temp;       /*gives warning*/
}
```

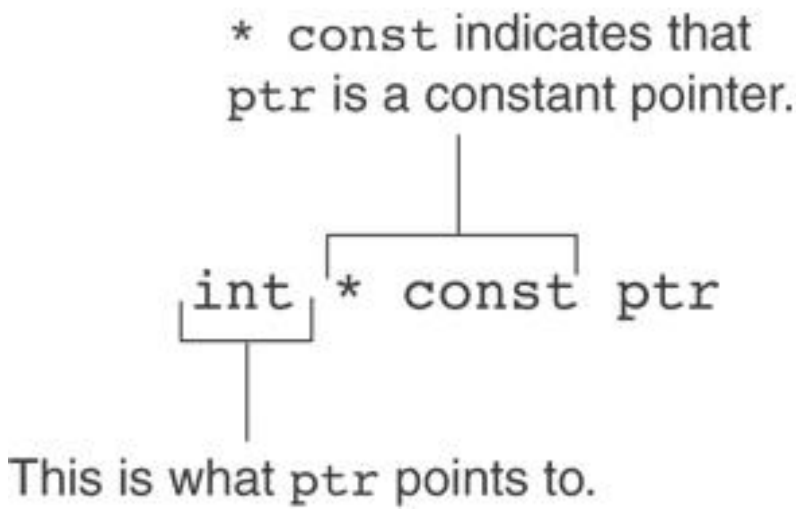
Constant Pointers

- 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

```
int num1 = 3;
int num2 = 7;
int *const ptr1 = &num1;
/*cannot change address in ptr1*/
int *ptr2 = &num2;
num1++;        /*no warning*/
(*ptr1)++;     /*no warning*/
ptr1 = ptr2; /*gives warning*/
```

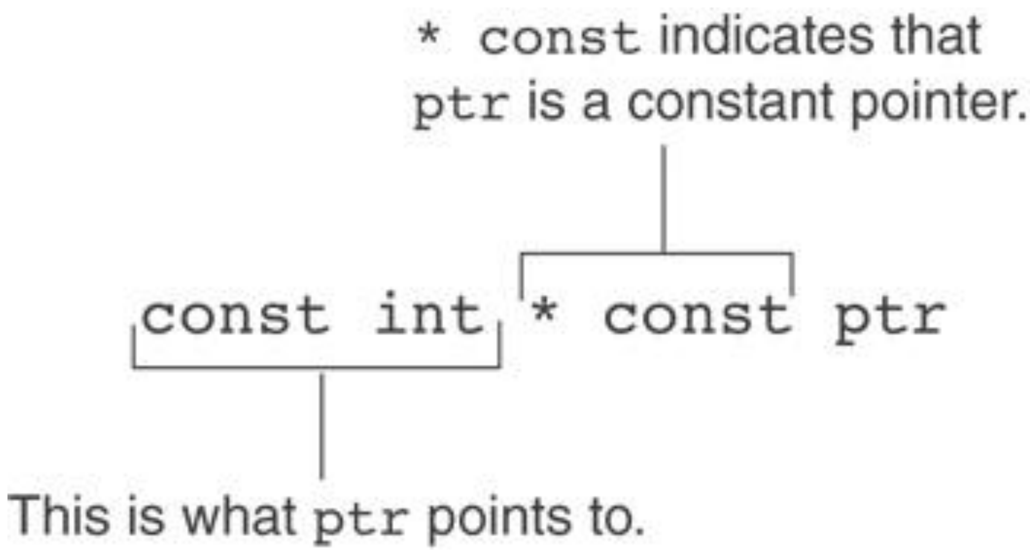
Constant Pointers to Constants

- 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



Constant Pointers

```
int num1 = 3, num2 = 7;
const int *const ptr1 = &num1;
/*cannot change address and   cannot
  change value at that address*/
int *ptr2 = &num2;
num1++;         /*no warning*/
(*ptr1)++;      /*gives warning*/
ptr1 = ptr2; /*gives warning*/
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

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

Character Pointers

- `char array[] = "this is an array";`
  - Array of characters + '\0'
  - Can change the characters in the array
  - Cannot change the array’s address
- `char *ptr = "this is a pointer";`
  - Points to constant characters + '\0'
  - Cannot change the string constant
  - Can change the pointer value

Character Pointers

```
#include <stdio.h>
int main(){
   char a[] = "character array";
   char *b  = "constant array";
   a=b; /*compiler error: incompatible types
          in assignment*/
   a[0]='C';
   b[0]='A'; /*runtime error: Segmentation
               Fault*/
   return 0;
}
```

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.

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

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;
}
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

A Student Structure

```
struct Student
{
   int studentID;
   string name;
   short yearInSchool;
   double gpa;
};
 Student stu1;
 cin >> stu1.studentID;
 getline(cin, stu1.name);
 stu1.gpa = 3.75;
```

structure tag

structure members

Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use `&` operator to assign address:

```
stuPtr = &stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

- Must use () to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate () and use clearer notation:

```
cout << stuPtr->studentID;
```

From Program 11-9

```
void getData(Student *s)
{
   // Get the student name.
   cout << "Student name: ";
   cin.getline(s->name, NAME_LENGTH);

   // Get the student ID number.
   cout << "Student ID Number: ";
   cin >> s->idNum;

   // Get the credit hours enrolled.
   cout << "Credit Hours Enrolled: ";
   cin >> s->creditHours;

   // Get the GPA.
   cout << "Current GPA: ";
   cin >> s->gpa;
}
```

Outline

- What is a Pointer?
- Arrays vs. Pointers
- Pointer Arithmetic
- Initializing Pointers
- Comparing Pointers
- Pointer As Function Parameters
- Dynamic Memory Allocation
- Return Pointers from Functions
- Pointers to structures
- 2-dim Dynamic Arrays

Two-Dimensional Arrays

- Visually like a matrix with rows and columns
- Two big questions to answer
  - How to allocate a 2-dim array?
  - How to de-allocate a 2-dim array?