# Pointer & Reference Variable

# Pointers and the Address Operator

- Each variable in a program is stored at a unique address in memory

- Use the address operator **&** to get the address of a variable:

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

- The dereference operator (*) allows us to access the value at a particular address:

```
cout << *&num
```

- The address of a memory location is a pointer

# Pointer Variables

- Pointer variable (pointer): variable that *holds an address* as its value.
- Pointers provide an alternate way to access memory locations
- Because a pointer variable holds the address of another piece of data, it "points" to the data

# 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"

- Spacing in definition does not matter:

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

# Pointer Variables

- Assignment:

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

- Memory layout:


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

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

```
cout << intptr;  // prints 0x4a00
cout << *intptr; // prints 25
```

**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 Dereference Operator

- The (`*`) 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 Output**

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

# Reusing a pointer

- When we apply the indirection operator to the pointer `ptr`, we are not working with `ptr` as and address but with the item to which `ptr` points

- Thus, we can reuse `ptr` to **point to different variables** and make changes to the value of each variable

# Reusing a pointer

```
int main()
{
   int x = 25, y = 50, z = 75;  // Three int variables
   int *ptr;                    // Pointer variable

   // Display the contents of x, y, and z.
   cout << "Here are the values of x, y, and z:\n";
   cout << x << " " << y << " " << z << endl;

   // Use the pointer to manipulate x, y, and z.
   ptr = &x;      // Store the address of x in ptr.
   *ptr += 100;   // Add 100 to the value in x.
   ptr = &y;      // Store the address of y in ptr.
   *ptr += 100;   // Add 100 to the value in y.
   ptr = &z;      // Store the address of z in ptr.
   *ptr += 100;   // Add 100 to the value in z.

   // Display the contents of x, y, and z.
   cout << "Once again, here are the values of x, y, and z:\n";
   cout << x << " " << y << " " << z << endl;
   return 0;
}
```

OUTPUT:

```
Here are the values of x, y, and z:
25 50 75
Once again, here are the values of x, y, and z:
125 150 175
```

# 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
```

# Arrays & 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};
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;`<br>`(vals + 2)= address of 3rd element` |

# Array Access

- Array notation

```
vals[i]
```

is equivalent to the pointer notation

```
*(vals + i)
```

- No bounds checking performed on array access

```
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
```

# Array Names vs Pointer Variables

- The only difference between array names and pointer variables is that you can not change the address an array name points to
  - Array names are ***pointer constants***

```
double readings [20], totals[20]
double *dprt;
```

| LEGAL: | ILLEGAL: |
|---|---|
| `dptr = readings;` | `readings = totals;` |
| `dptr = totals;` | `totals = dptr;` |

# 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 - vals;
```

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

# Pointer Arithmetic

```
int vals[]={4,7,11};
int *valptr = vals;
cout<<"\nvalptr++: "<<valptr++;
cout<<" *valptr: "<<*valptr<<endl;

cout<<"valptr--: "<<valptr--;
cout<<" *valptr: "<<*valptr<<endl;

cout <<"*(valptr+2): "<< *(valptr + 2)<<endl;

valptr = vals;
valptr+=2;
cout<<"valptr+=2: "<<valptr<<endl;

int y = valptr-vals;
cout<<"valptr: "<<valptr<<" vals: "<<vals;
cout<<" *valptr: "<<*valptr<<" *vals: "<<*vals<<endl;
cout<<" valptr minus vals: "<<y<<endl;
```

```
valptr++: 0x22ff20 *valptr: 7

valptr--: 0x22ff24 *valptr: 4

*(valptr+2): 11


valptr+=2: 0x22ff28


valptr: 0x22ff28
vals: 0x22ff20
*valptr: 11 *vals: 4

valptr minus vals: 2
```

```
    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
```

# Pointer Arithmetic - Summary

| **Operation** | **Example**<br>`int vals[]={4,7,11};`<br>`int *valptr = vals;` |
|---|---|
| `++, --` | `valptr++; // points at 7`<br>`valptr--; // now points at 4` |
| `+, -` (pointer and `int`) | `cout << *(valptr + 2); // 11` |
| `+=,-=` (pointer and `int`) | `valptr = vals; // points at 4`<br>`valptr += 2;   // points at 11` |
| `-` (pointer from pointer) | `cout << valptr-val; // difference`<br>`//(number of ints) between valptr`<br>`// and val` |

# Pointer Initialization

- Just like normal variables, pointers are not initialized when they are declared.
- Unless a value is assigned, a pointer will point to some **garbage** address by default.
- Besides memory addresses, there is one additional value that a pointer can hold: a **null** value.
- A **null value** is a special value that means the pointer is not pointing at anything. A pointer holding a null value is called a **null pointer**.

```
float *ptr { 0 };  // ptr is now a null pointer

float *ptr2; // ptr2 is uninitialized
ptr2 = 0; // ptr2 is now a null pointer
```

# Null pointers

- Pointers convert to boolean false if they are null, and boolean true if they are non-null. Therefore, we can use a conditional to test whether a pointer is null or not:

```
double *ptr { 0 };

// pointers convert to boolean false if they are null, and boolean true if they are non-null
if (ptr)
    cout << "ptr is pointing to a double value.";
else
    cout << "ptr is a null pointer.";
```

- ***Best practice:*** *Initialize your pointers to a null value if you're not giving them another value*.

- Dereferencing a garbage pointer would lead to undefined results. Dereferencing a null pointer also results in undefined behavior. In most cases, it will *crash your application*.

# 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
```

- Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```

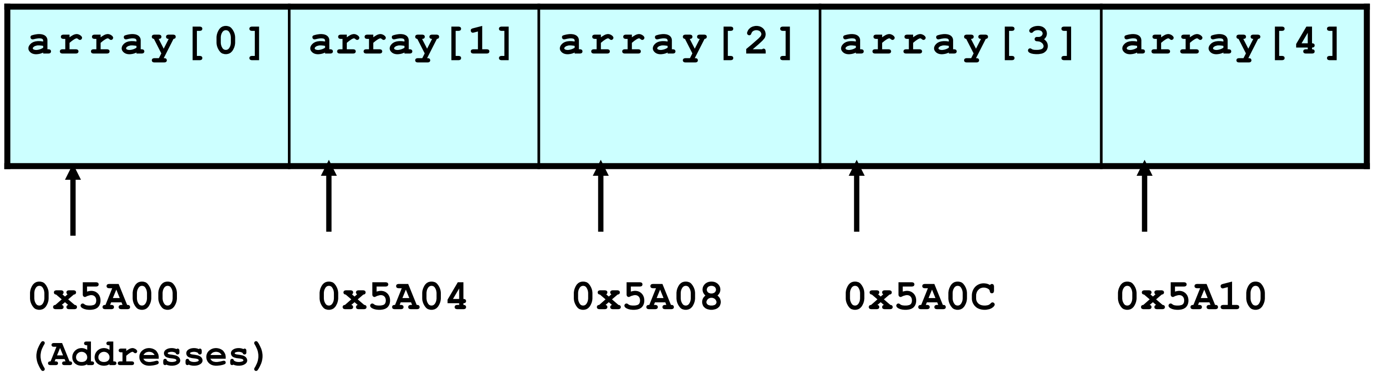
# Comparing Pointers

- 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
```

# Comparing Pointers

- Addresses grow larger for each subsequent element in an array



# 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;
*dptr = 56.78; //assign value
total +=*dptr; // use in computation
```

- `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 *throw an exception* and terminate if not enough memory available to allocate

# Releasing Dynamic Memory

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

```
delete dptr;
```

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

```
delete [] arrayptr;
```

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

- The delete operator does not *actually* delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later).
- Although it looks like we're deleting a *variable*, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value just like any other variable.

**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
   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];
   }

```

```
33        // Calculate the total sales
34        for (count = 0; count < numDays; count++)
35        {
36           total += sales[count];
37        }
38
39        // Calculate the average sales per day
40        average = total / numDays;
41
42        // Display the results
43        cout << fixed << showpoint << setprecision(2);
44        cout << "\n\nTotal Sales: $" << total << endl;
45        cout << "Average Sales: $" << average << endl;
46
47        // Free dynamically allocated memory
48        delete [] sales;
49        sales = 0;        // Make sales point to null.
50
51        return 0;
52     }
```

**Program Output with Example Input Shown in Bold**

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

Total Sales: $4067.13<br>
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.

# 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

# Pointers to Constants and Constant Pointers

- Pointer to a constant: cannot change the value that is pointed at
- Constant pointer: address in pointer cannot change once pointer is initialized
- To pass the address of a **`const`** item into a pointer, the pointer must be defined as a pointer to a **`const`** item
  - What is the purpose of a **`const`** item?

# Pointers to Constant

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

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

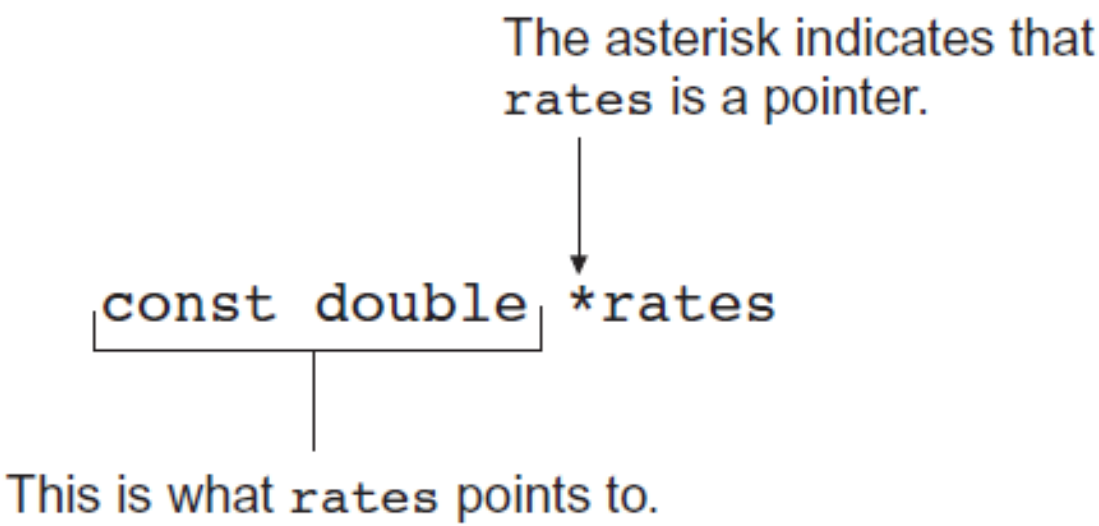
# Pointers to Constants

- 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 };
double overtimeRates[SIZE] =
     { 18.55, 17.45, 12.85,
       14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an **array of constant doubles** and `overtimeRates` is an array of non-constant doubles

# Pointer to Constant – What does the Definition Mean?



# Pointer That Is a Constant Pointer

`* const` indicates that `ptr` is a constant pointer.

```
int * const ptr
```

This is what `ptr` points to.

# Constant Pointers to Constants

`* const` indicates that `ptr` is a constant pointer.

```
const int * const ptr
```

This is what `ptr` points to.

# Pointer Types

- **`const double * rates`** A pointer to a **`const`** points to a constant item
  - The data that the pointer points to can not change, but the pointer itself can change
- **`int * const ptr`** With a **`const`** pointer, the pointer itself is constant.
  - Once the pointer is initialized with an address, it can not point to anything else but the contents of what it points to can change
- **`const int * const ptr`** With a **`const`** pointer to a **`const`**, can not use **`ptr`** to change the contents of **`value`**

# Constant Pointers

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

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

- Must be initialized when defined

- While the <u>address</u> in the pointer cannot change, the <u>data</u> at that address may be changed

# Constant Pointers

- A constant pointer, **`ptr`**, is a pointer that is initialized with an address, and cannot point to anything else.
- We can use **`ptr`** to change the contents of **`value`**
- Example

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

# Regular Pointers

```
int value2= 987;
int value3=5678;
int * ptr2 = &value2;
cout<<"\nRegular Pointer\n";
cout <<"value2: "<<value2<<" ptr2: "<<ptr2<<" *ptr2: "<<*ptr2<<endl;
*ptr2=12345;
cout <<"value2: "<<value2<<" ptr2: "<<ptr2<<" *ptr2: "<<*ptr2<<endl;
//LEGAL WITH REGUALR POINTER, CHANGE OF ADDRESS
ptr2= &value3;
cout <<"value3: "<<value3<<" ptr2: "<<ptr2<<" *ptr2: "<<*ptr2<<endl;
```

```
Regular Pointer
value2: 987 ptr2: 0x22ff0c *ptr2: 987
value2: 12345 ptr2: 0x22ff0c *ptr2: 12345
value3: 5678 ptr2: 0x22ff08 *ptr2: 5678
```

# Constant Pointers

```
int value = 22;
int value4 = 99;
int * const ptr = &value;
cout<<"\nConstant Pointer\n";
cout <<"value: "<<value<<" ptr: "<<ptr<<" *ptr: "<<*ptr<<endl;
value=99;
cout <<"value: "<<value<<" ptr: "<<ptr<<" *ptr: "<<*ptr<<endl;
*ptr=12345;
cout <<"value: "<<value<<" ptr: "<<ptr<<" *ptr: "<<*ptr<<endl;
// THIS CAN NOT BE DONE, CAN NOT CHANGE ADDRESS THAT ptr POINTS TO
// ptr = &value4;
```

```
Constant Pointer
value: 22 ptr: 0x22ff18 *ptr: 22
value: 99 ptr: 0x22ff18 *ptr: 99
value: 12345 ptr: 0x22ff18 *ptr: 12345
```

# Pointer to a Constant

```
int value6 = 666;
int value7 = 777;
const int * ptr6 = &value6;
cout<<"\nPointer to a Constant\n";
cout <<"value6: "<<value6<<" ptr6: "<<ptr6<<" *ptr6: "<<*ptr6<<endl;
value6=6666;
cout <<"value6: "<<value6<<" ptr6: "<<ptr6<<" *ptr6: "<<*ptr6<<endl;
ptr6=&value7;
cout <<"value7: "<<value7<<" ptr6: "<<ptr6<<" *ptr6: "<<*ptr6<<endl;
//CAN NOT BE DONE
//   *ptr6=6543;
```

```
Pointer to a Constant
value6: 666 ptr6: 0x22ff74 *ptr6: 666
value6: 6666 ptr6: 0x22ff74 *ptr6: 6666
value7: 777 ptr6: 0x22ff70 *ptr6: 777
```

# 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 Pointer to a Constant

```
int value5 = 55;
int value12 = 1212;
const int * const ptr5 = &value5;
cout<<"\nConstant Pointer to a Constant\n";
cout <<"value5: "<<value5<<" ptr5: "<<ptr5<<" *ptr5: "<<*ptr5<<endl;
value5=777;
cout <<"value5: "<<value5<<" ptr5: "<<ptr5<<" *ptr5: "<<*ptr5<<endl;
//CAN NOT PERFORM EITHER ONE OF THESE OPERATIONS
//    ptr5 = &value12;
//    *ptr5=6543;
```

```
Constant Pointer to a Constant
value5: 55 ptr5: 0x22ff74 *ptr5: 55
value5: 777 ptr5: 0x22ff74 *ptr5: 777
```

# Pointer to Structure

# Pointers to Structures

- Can create pointers to objects and structure variables

```
struct Student {
    int studentID;
    char* address;
    string phone;
};
Student stu1;
Student *stuPtr = &stu1;
```

- Need `()` when using `*` and `.`

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

as the dot selector has higher priority than the * operator

# Structure Pointer Operator

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

```
stuPtr->studentID = 12204;
```

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

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

# Selecting Members of Objects

| Expression | Meaning |
|---|---|
| **`stuPtr->studentID`** | Access the student. This is the same as **`(*stuPtr).grades`** |
| **`*(testPtr->address)`** | Access the value pointed at by **`testPtr->address`**. This is the same as **`*((*testPtr).address)`** |

# Dynamic Memory with Structure

- Can allocate dynamic structure variables and objects using pointers:

```
Student * stuPtr = new Student;
```

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

```
delete stuPtr ;
```

# Stopping Memory Leaks

- When using DAM, make sure that each call to **`new`** is eventually followed by a call to **`delete`** that frees the allocated memory and returns it to the heap.

- If not, memory leaks will occur in which the program loses track of dynamically allocated storage and never calls delete to free the memory

# Arrays of Pointers

- This example will have an array **`donations`** of unsorted integers
- An array of pointers is defined so that each element of **`donations`** is pointed to by an element of the array **`arrPtr`**
- We will then sort **`arrPtr`** based on the value stored and then display the sorted values without actually sorting the array **`donations`**

# Example: Dynamically Growing Arrays

- Write a program that starts out with an integer array of maximum size 5
- Values for the array are entered at the keyboard, one by one. When the number of values entered exceeds maximum size, create a new array with twice the maximum size, move the old values into the new array, save the last value entered and continue prompting for more values
- If the new maximum size is exceeded, follow the same steps as before

# Dynamically Growing Arrays

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

int main(){
    int *iptr, *iptrold;
    int num, maxsize=5;
    iptr = new int[maxsize];
    int count=0;

    cout<<"\nEnter a number: ";
    cin>>num;

 while ( !cin.eof() )
    {
        if (count < maxsize)
        {
            iptr[count] = num;
            count++;
        }
        else
        {
            cout<<"\nReached maxsize"<<endl;
            iptrold=iptr;
            iptr = new int[maxsize*2];
                for (int i=0;i<maxsize;i++)
                    *(iptr+i) = *(iptrold+i);
            delete  iptrold;
            iptr[count] = num;
            count++;
            maxsize=maxsize*2;
        }

        cout<<"\nEnter a number: ";
        cin>>num;
    }

    for (int i=0;i<maxsize;i++)
    {
        cout<<"iptr["<<i<<"]="<<iptr[i]<<endl;
    }
}
```

# Dynamically Growing Arrays

- Initialize two pointers

```
int *iptr, *iptrold;
```

- This code adds a new value to the array pointed to by iptr

```
If (count < maxsize)
{
    iptr[count] = num;
    count++;
}
```

# Dynamically Growing Arrays

- When the current size is exceeded
  - set iptrold to point to iptr
  - create a new array twice the size of maxsize pointed to by iptr
  - copy the values from iptrold to iptr
  - delete allocated memory where iptrold points to
  - add the last value read in to the new array
  - increase count by 1
  - double maxsize

```
else
 {
     cout<<"\nReached maxsize"<<endl;
     iptrold=iptr;
     iptr = new int[maxsize*2];
         for (int i=0;i<maxsize;i++)
             *(iptr+i) = *(iptrold+i);
     delete iptrold;
     iptr[count] = num;
     count++;
     maxsize=maxsize*2;
 }
```

# Dynamically Growing Arrays

```
Enter a number: 1

Enter a number: 2

Enter a number: 3

Enter a number: 4

Enter a number: 5

Enter a number: 6

Reached maxsize
maxsize: 5

Enter a number: 7

Enter a number: 8

Enter a number: 9

Enter a number: 10

Enter a number: 11

Reached maxsize
maxsize: 10
```

```
maxsize: 10

Enter a number: 12

Enter a number: 13

Enter a number: 14

Enter a number: 15

Enter a number: 16

Enter a number: 17

Enter a number: 18

Enter a number: 19

Enter a number: 20

Enter a number: 21

Reached maxsize
maxsize: 20
```

```
Enter a number: 22

Enter a number: 23

Enter a number: ^Z
iptr[0]=1
iptr[1]=2
iptr[2]=3
iptr[3]=4
iptr[4]=5
iptr[5]=6
iptr[6]=7
iptr[7]=8
iptr[8]=9
iptr[9]=10
iptr[10]=11
iptr[11]=12
iptr[12]=13
iptr[13]=14
iptr[14]=15
iptr[15]=16
iptr[16]=17
iptr[17]=18
iptr[18]=19
iptr[19]=20
iptr[20]=21
iptr[21]=22
iptr[22]=23
```

```
iptr[23]=0
iptr[24]=0
iptr[25]=0
iptr[26]=0
iptr[27]=0
iptr[28]=0
iptr[29]=0
iptr[30]=0
iptr[31]=0
iptr[32]=0
iptr[33]=0
iptr[34]=0
iptr[35]=0
iptr[36]=0
iptr[37]=0
iptr[38]=0
iptr[39]=0
```

# Reference Variables

# Reference Variables

- A reference (to a non-const value) is declared by using an ampersand (&) between the reference type and the variable name:

- int value = 5;            // normal integer
- int &ref = value;       // reference to variable value

- In this context, the ampersand does not mean “address of”, it means “reference to”.
- References to non-const values are often just called “references” for short.

# References as aliases

- References generally act identically to the values they're referencing. In this sense, a reference acts as an alias for the object being referenced.

```
#include <iostream>

int main()
{
    int value = 5; // normal integer
    int &ref = value; // reference to variable value

    value = 6; // value is now 6
    ref = 7; // value is now 7

    std::cout << value; // prints 7
    ++ref;
    std::cout << value; // prints 8

    return 0;
}
```

This code prints:

```
7
8
```

# References as aliases

- In the above example, ref and value are treated synonymously.
- Using the address-of operator on a reference returns the address of the value being referenced:

```
cout << &value; // prints 0012FF7C
cout << &ref; // prints 0012FF7C
```

# References must be initialized

- References must be initialized when created:

```
int value = 5;
int &ref = value; // valid reference, initialized to variable value

int &invalidRef; // invalid, needs to reference something
```

- References to non-const values can only be initialized with non-const l-values. They can not be initialized with const l-values or r-values.

```
int x = 5;
int &ref1 = x; // okay, x is an non-const l-value

const int y = 7;
int &ref2 = y; // not okay, y is a const l-value

int &ref3 = 6; // not okay, 6 is an r-value
```

# References can not be reassigned

- Once initialized, a reference can not be changed to reference another variable.

```
int value1 = 5;
int value2 = 6;

int &ref = value1; // okay, ref is now an alias for value1
ref = value2; // assigns 6 (the value of value2) to value1 -- does NOT change the reference!
```

- Note that the second statement may not do what you might expect! Instead of reassigning ref to reference variable value2, it instead assigns the value from value2 to value1 (which ref is a reference of).

# References and Const

- Unlike references to non-const values, which can only be initialized with non-const l-values, references to const values can be initialized with **non-const l-value**, **const l-values**, and **r-values**.

```
int x = 5;
const int &ref1 = x; // okay, x is a non-const l-value

const int y = 7;
const int &ref2 = y; // okay, y is a const l-value

const int &ref3 = 6; // okay, 6 is an r-value
```

# Member selection with pointers and references

- It is common to have either a pointer or a reference to a struct (or class).

```cpp
struct Person
{
    int age;
    double weight;
};
Person person;

// Member selection using actual struct variable
person.age = 5;
```

- This syntax also works for references:

```cpp
struct Person
{
    int age;
    double weight;
};
Person person; // define a person

// Member selection using reference to struct
Person &ref = person;
ref.age = 5;
```

# Member selection with pointers and references

- With a pointer, you need to dereference the pointer first:

```
struct Person
{
    int age;
    double weight;
};
Person person;

// Member selection using pointer to struct
Person *ptr = &person;
(*ptr).age= 5;
```

- Or, you can choose to use the member selection operator (**->**)

```
(*ptr).age  = 5;
ptr->age = 5;
```

- This is not only easier to type, but is also much less prone to error