Lecture 1 – part 2

Pointers, structs, and classes

Pointers and Addresses

- 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;
```

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

- 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

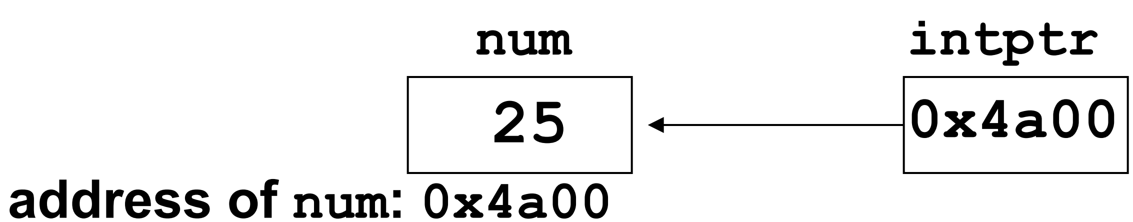
Pointer Variables

- Definition and assignment:

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

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

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



Pointers and Arrays

- An array name is the starting address of the array

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

  cout << vals;   // displays 0x4a00

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

- An array name can be used as a pointer constant

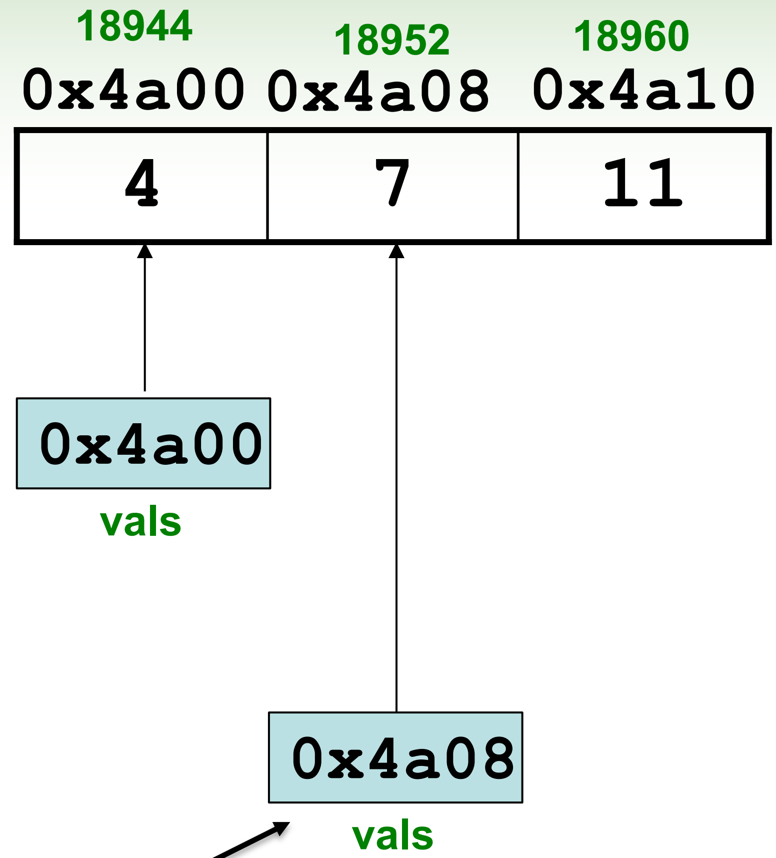
  int vals[] = {4, 7, 11};

  cout << *vals;   // displays 4

- A pointer can be used as an array name

  int *valptr = vals;

  cout << valptr[1]; // displays 7



- 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**

Arrays Access

- Array notation

  vals[i]

is equivalent to the pointer notation

  ***(vals + i)**

- Remember that no bounds checking is performed on array access

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;

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
```

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

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 val[ISIZE], *valptr = val;
```

- The initial value must have the correct type

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

- In C++ 11, putting empty { } after a variable definition indicates that the variable should be initialized to its default value
- C++ 11 also has the the key word nullptr to indicate that a pointer variable does not contain a valid memory location

- You can use

```
int *ptr = nullptr;
```

or

```
int *ptr{ };
```

- 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
```


Pointers as Function Parameters

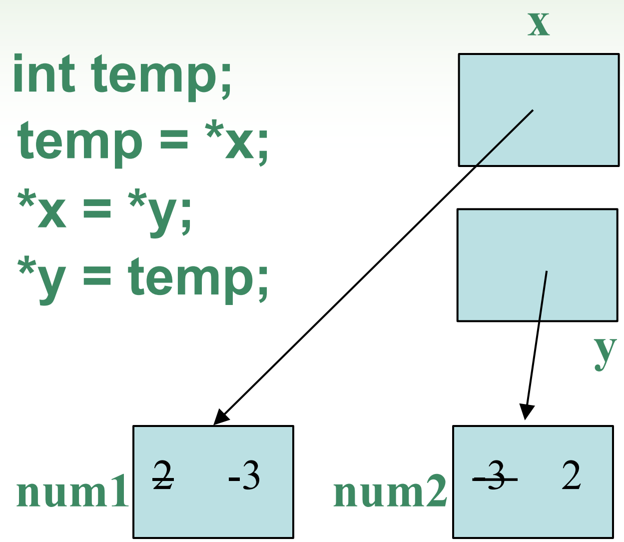
- A pointer can be a parameter
- It works like a reference parameter to allow changes to argument from within a function
- A pointer parameter must be explicitly dereferenced to access the contents at that address
- Requires:
  - asterisk * on parameter in prototype and header

    `void getNum(int *ptr);`
  - asterisk * in body to dereference the pointer

    `cin >> *ptr;`
  - address as argument to the function in the call

    `getNum(&num);`

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




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

Passing Arrays as Pointers

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

void print(int *p, int size){
    for (int i=0;i<size;i++)
        cout << p[i] << ":";
    cout << endl;
}

void print1(int *p, int size){
    for (int i=0;i<size;i++)
        cout << *(p+i) << ":";
    cout << endl;
}
```

```
int main(int argc, const char * argv[]) {
    int a[] = {1,2,3,4,5};
    print (a,5);
    print1 (a,5);
    return 0;
}
```

```
1:2:3:4:5:
1:2:3:4:5:
```

Pointers and the const keyword

The asterisk indicates that `rates` is a pointer.

```
const double *rates
```

This is what `rates` points to.

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

```
int * const ptr
```

This is what `ptr` points to.

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

```
const int * const ptr
```

This is what `ptr` points to.

Dynamic Memory Allocation

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

int main(int argc, const char * argv[ ]) {
    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];
    }
    delete count;
    delete [ ] arrayptr;
    return 0;
}
```

- 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

Smart Pointers

- Introduced in C++ 11
- They can be used to solve the following problems in a large software project
  - dangling pointers – pointers whose memory is deleted while the pointer is still being used
  - memory leaks – allocated memory that is no longer needed but is not deleted
  - double-deletion – two different pointers de-allocating the same memory
- Smart pointers are objects that work like pointers.
- Unlike regular (raw) pointers, smart pointers can automatically delete dynamic memory that is no longer being used.
- There are three types of smart pointers:
  - unique pointers (**unique_ptr**)
  - shared pointers (**shared_ptr**)
  - weak pointers (**weak_ptr**)

Unique Pointers

- A smart pointer owns (or manages) the object that it points to.
- A unique pointer points to a dynamically allocated object that has a single owner.
- Ownership can be transferred to another unique pointer.
- Memory for the object is deallocated when the owning unique pointer goes out of scope, or if it takes ownership of a different object.
- Requires the **<memory>** header file
- Create a unique pointer that points to an int:

```
unique_ptr<int> uptr(new int);
```

- Assign the value 5 to it and print it:

```
*uptr = 5;
cout << *uptr;
```

- Transfer ownership to unique pointer ptr2:

```
unique_ptr<int> uptr2;
uptr2 = move(uptr);
```

Unique Pointers - move

- In a statement such as:

  uptr2 = move(uptr);

  - Any object owned by **uptr2** is deallocated
  - **uptr2** takes ownership of the object previously owned by **uptr**
  - **uptr** becomes empty
- The move() function is required on the argument when passing a unique pointer by value.
- The move() function is not required for pass by ref
- A unique pointer can be returned from a function, as the compiler automatically uses move() in this case.
- Unique pointers deallocate the memory for their objects when they go out of scope.
- To manually deallocate memory, use

  uptr = nullptr; or **uptr.reset();**
- Use array notation when using an unique pointer to allocate memory for an array

  unique_ptr<int[]>uptr3(new int[5]);

- Doing so ensures that the proper deallocation (delete[] instead of delete) will be used.

Shared Pointers

- A smart pointer owns (or manages) the object that it points to.
- A shared pointer points to a dynamically allocated object that may have multiple owners.
- A control block manages the reference count of the number of shared owners and also possibly the raw pointer if one exists.
- Create a shared pointer to point to an existing dynamic object declared with a raw pointer:

```
int * rawPtr = new int;

shared_ptr<int> uptr4(rawPtr);
```

- Create a second shared pointer initialized to the same object:

```
shared_ptr<int>uptr5 = uptr4;
```

- **rawPtr**, **uptr4**, and **uptr5** are all tracked in the control block.

Shared Pointers

- Be careful that all references to a dynamic object are tracked in the same control block
- In the code below:

```
int * rawPtr = new int;
shared_ptr<int> uptr4(rawPtr);
shared_ptr<int> uptr5(rawPtr);
```

- Two control blocks are created. This can cause a dangling pointer.
- Creating a shared pointer involves memory for the object and memory for the control block.
- These memory allocations can be combined by using the make_shared function:

```
shared_ptr<int> uptr6 = make_shared<int>();
```

- You can also pass parameters to a constructor using an overloaded version of make_shared.

Shared Pointers

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

void fun(shared_ptr<int> ptr){
    shared_ptr<int> ptr3 = ptr;
    cout << ptr3.use_count()
         << endl;
}
```

```
int main()
{
    shared_ptr<int> ptr1,ptr2;
    ptr1 = make_shared<int>(10);

    ptr2 = ptr1;
    *ptr2 = 100;

    cout << *ptr2 << endl;
    cout << *ptr1 << endl;

    cout << ptr2.use_count() << endl;
    ptr1 = nullptr;
    cout << ptr2.use_count() << endl;

    fun(ptr2);
    cout << ptr2.use_count() << endl;
    return 0;
}
```

```
100
100
2
1
3
1
```