Pointers and Dynamic Arrays

- Pointer Variables
- C++ Memory Management
- Dynamic Variables
- Pointers as Parameters
- Arrays as Parameters
- Prescription for a Dynamic Class
- BagDynamic Class
- C++ 11 Smart Pointers

Pointer Variables

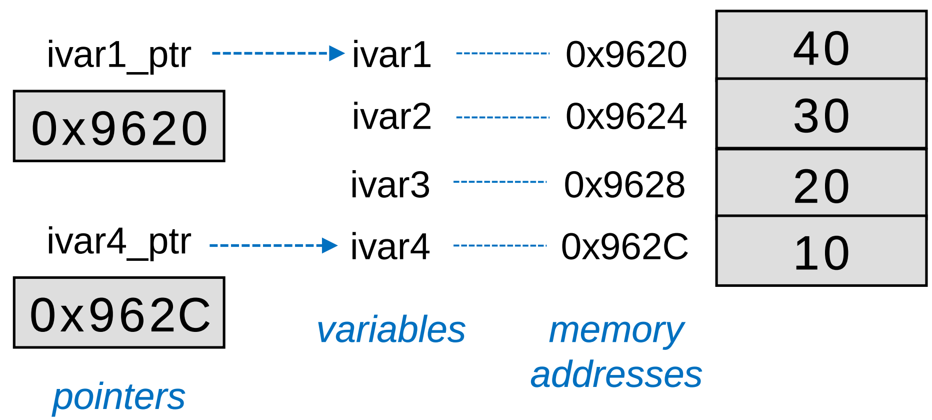
- Pointer → variable that contains memory address of another variable
  - Each byte in memory has an address
- Declaration format:
  - `data_type *var1_ptr, *var2_ptr;`
  - `data_type* var_ptr;`
- Use address operator ‘&’ to get the memory address of variable

Pointer Variables

- Use dereferencing operator '*' with pointer variable to dereference address and access memory location
- Cursor → pointer variable that accesses all items in data container

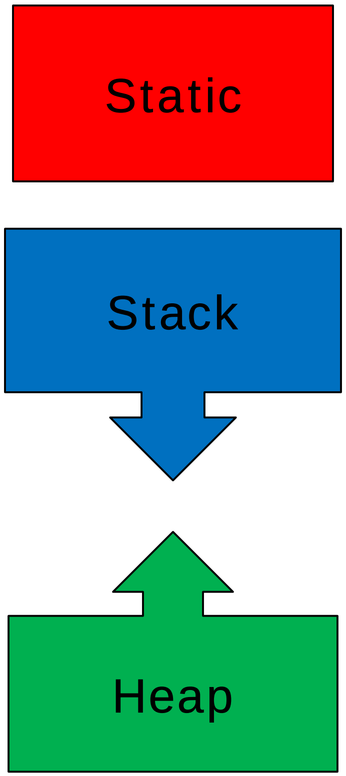
```
int ivar1 = 40, ivar2 = 30,
   ivar3 = 20, ivar4 = 10;
int *ivar1_ptr = &ivar1,
   *ivar4_ptr = nullptr;
ivar4_ptr = &ivar4;
cout << ivar1 << endl;
cout << *ivar1_ptr << endl;

?? ivar1_ptr < ivar4_ptr   ??
?? *ivar1_ptr < *ivar4_ptr ??
```



C++ Memory Management

- Determined at compile time
  - Static
    - Global/static variables and constants
  - Stack
    - Variables local to function
    - LIFO structure optimized by CPU
- Determined at run time
  - Heap
    - Dynamic memory allocation/release
    - Managed by programmer



Dynamic Variables

- Are not declared, but created during program execution
- ‘new’ operator → allocates, and potentially initializes, memory
  - Returns pointer to allocated memory, or
  - Throws bad_alloc exception if no heap memory available
  - ‘nothrow’ format returns nullptr upon failure
    - NULL for pre C++ 11.0 compilers

Dynamic Variables

- Allocation format:
  - `data_type *ptr1 = new data_type;`
  - `data_type *ptr2 = new data_type(init_value);`
  - `data_type *arr_ptr = new data_type[array_size]`
- Efficient to return memory to heap when no longer needed
- ‘delete’ operator → deallocates, or releases, memory allocated with ‘new’
- Release format:
  - `delete ptr1;`
  - `delete ptr2;`
  - `delete [ ] arr_ptr;`

Pointers As Parameters

- Pointer as value parameter permits modification of *value to which pointer refers*
  - `void change_val(int* i_ptr);`
  - `void change_val(int *i_ptr);`
- Pointer as reference parameter permits modification of *pointer value*
  - `void change_ptr(int*& i_ptr);`

Arrays As Parameters

- Array parameters treated as pointer to first element of array
  - Size of array *should be* passed as separate parameter
    - `void double_vals(int *i_ptr, int size);`
    - `void double_vals(int i_ptr[], int size);`
- Array parameter can use index access

```
for (int i = 0; i < size; i++)
    i_ptr[i] = i_ptr[i] * 2;
```

Arrays As Parameters

- Array parameter can use pointer arithmetic

```
for (int i = 0; i < size; i++)
   *(i_ptr + i) = *(i_ptr + i) * 2;
```

- 'const' keyword prevents modification of array parameter elements
  - `void print_vals(const int *i_ptr, int size);`
  - `void print_vals(const int i_ptr[], int size);`

Prescription for a Dynamic Class

- Notes for a Dynamic Class
  - Some member variables are pointers
  - Member functions will allocate and release dynamic memory as needed
    - constructors, destructors, insert/add, copy (*when container to copy from is larger than current*)
  - Default copy constructor and assignment operator must be overridden to avoid invalid memory access

Prescription for a Dynamic Class: Copy Constructor

- Parameter of reference to class itself is known as copy constructor
  - Use const qualifier to prevent modification of parameter
  - Implicitly invoked by
    - initialization in variable declaration
      - `class_name c1, c2(c1), c3 = c2;`
    - pass by value parameter
      - `void display_class(class_name cvar);`
    - return value of function
      - `class_name get_class();`
  - If not provided, default copy constructor created
    - *simple* memberwise assignment

BagDynamic Class

- Rules for implementation
  - *Number* of items stored in member variable `used`
  - Bag *items* stored in partially filled dynamic array member variable `data`
  - Total *size* of dynamic array in member variable `capacity`

how much memory is allocated

BagDynamic Class

- Differences from BagFixed
  - Constructor with default parameter sets initial capacity
    - Include **`DEFAULT_CAPACITY`** static class constant
  - Examine bag capacity when item added (*via insert or +=*) and increase as necessary
  - Override automatic copy constructor and assignment operator
  - Member functions that can allocate dynamic memory
    - default constructor
    - reserve
    - insert
    - += operator
    - + operator
  - Destructor releases dynamic memory and returns it to the heap

C++ 11 Smart Pointers

- Automatically releases dynamically allocated memory when no longer used
  - class template initialized by raw pointer referencing heap allocated memory
    - memory managed through standard scoping rules of reference counters
    - cannot reassign pointer values
      - potential memory leak on heap
    - cannot directly used arithmetic operators to increment/decrement pointers
      - use of .get() to reference raw pointer will cause issues when memory automatically released

C++ 11 Smart Pointers

- Use header file:

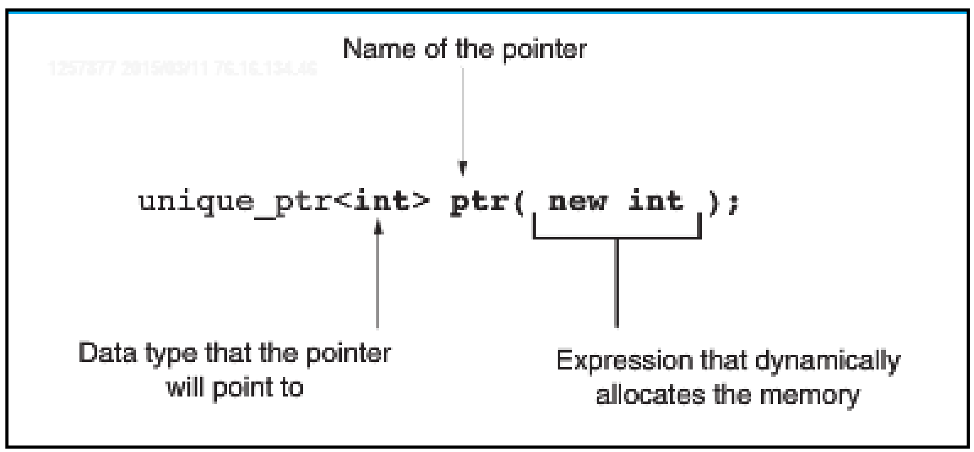
```
#include <memory>
```

- Types of smart pointers:
  - unique_ptr → only single reference
  - shared_ptr → possible multiple references
    - memory released when reference count is 0
  - weak_ptr → copy of shared_ptr that doesn't affect reference count

C++ 11 Smart Pointers

- Syntax:

```
unique_ptr<int> ptr(new int);
unique_ptr<int> iptr (new int(15));
unique_ptr<int[]> aptr (new int[size]);
```



C++ 11 Smart Pointers

```
unique_ptr<int[]> getRandArr(int numInts) {
  // create dynamic smart pointer array
  unique_ptr<int[]> sptr(new int[numInts]);
  // populate array with random numbers 1 to 100
  unsigned seed = static_cast<unsigned>(time(0));
  srand(seed);
  for (int index = 0; index < numInts; index++)
    sptr[index] = 1 + rand() % 100;
  // return pointer to array of random numbers
  return sptr;
}
```

```
unique_ptr<int[]> rArray = getRandArr(5);
for (int i = 0; i < 5; i++)
    cout << rArray[i] << " ";
cout << endl;
```

No need to worry about releasing memory!