Learning Objectives

- Pointers
  - Data type and pointer variables
  - Declaration and manipulation
  - Address versus dereference operators
  - Memory management
- Pointers to user-defined data type
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic
  - Multi-dimensional arrays

Memory Addresses

- Every byte in memory has an integer address
  - Ex: a computer with 256MB RAM
  - Addresses start from 0 to $256 \times 1024 \times 1024$
- An int variable uses 4 bytes
  - Ex: int a = 10;
  - Each address stores one byte
  - Store in address 0012FED4 to 0012FED7

Introduction to Pointers

- [Def] pointer:
  - Is a memory address of a variable
- Syntax:
  - `datatype * identifier;`
- Recall: memory divided
  - Numbered memory locations (index)
  - Addresses used as content of a pointer variable
- Don’t be panic! Seen this before.
  - Call-by-reference parameters
  - Address of actual argument was passed
**Declaring Pointer Variables**

- Pointers declared like other types
  - `*` before variable names
  - produce `pointer` to that type
- 3 forms are equivalent
  - `int *ptr;` /most suggested by textbook
  - `int ptr;` /most convenient practically
  - `int *ptr;`
  - `*` must be located before each variable
- Example: `int *ptr1, var1, *ptr2, var2;`
  - `ptr1, ptr2` hold pointers to `int` variables
  - `var1, var2` are ordinary `int` variables

---

**Pointer Variables**

- Pointers can be viewed as one *datatype*
  - can store pointers in variables
  - not `int, double, char` and etc.
  - instead, a *pointer (or an arrow)* to `int, double, char` and etc.
- Example: `double * p;`
  - `p` is declared a *pointer-to-double* variable
  - can hold pointers to variables of type `double`
  - but not other types like `char`
- EX: `sizeof(p)=8` on x86_64 workstation.

---

**Addresses and Numbers**

- Content of a pointer variable is an address
  - address is an *integer* number (index for memory location)
  - recall `sizeof(double*)`
- But pointer is actually NOT typed in integer
  - not in `int` datatype
  - not crazy ⇒ due to *abstraction*
- C++ forces pointers be used as addresses
  - cannot be used as numbers
  - even though it is a number

---

**Dereferencing Operator `*`**

- Two roles of `*` in C/C++:
  - binary multiplication operator, `ex: 8*5`
  - a unary operator, `ex: int* iptr;`
- As `*` is used as a unary operator,
  - called dereferencing operator (or *indirection* operator)
  - refer to object to which its operand (that is, a pointer) points

```
iptr = [4000]  
8
```
**Address Operator: &**

- &: the "address of" operator
  - also used to specify call-by-reference parameters \( \Rightarrow \) not coincident
- Recall: call-by-reference parameters pass address of the actual argument
- Example:
  ```
  void func(int& var2);
  ...
  func(var1); //int& var2 = var1;
  \Rightarrow \text{var2 and var1 will have the same address}
  ```
- Operator's two uses are closely related

**Pointing To ... (1/2)**

- Terminology, view
  - talk of pointing, not addresses
  - pointer variable points to ordinary variable
  - leave address talk out
- Example:
  ```
  int *p1, *p2, v1, v2;
  p1 = &v1;
  ... set pointer variable p1 to point to int variable v1
  ... or p1 is assigned the address of v1
  ```

**Pointing to ... (2/2)**

- Example:
  ```
  int *p1, *p2, v1, v2;
  p1 = &v1;
  ```
- Two ways to refer to v1 now:
  - variable v1 itself:
    ```
    cout << v1;
    ```
  - via pointer p1:
    ```
    cout << *p1;
    ```
- Dereference operator, *
  - pointer variable dereferenced
  - mean: retrieve meta that p1 points to

**Another Example**

- Consider:
  ```
  int v1 = 0;
  int* p1;
  p1 = &v1;
  *p1 = 45;
  cout << v1 << endl;
  cout << *p1 << endl;
  ```
- Produces output:
  ```
  45
  45
  ```
- p1 and v1 refer to same variable
Assignment of Pointers (1/2)

- Pointer variables can be assigned:
  ```
  int *p1, *p2;
p1 = p2; //ex: address of p2 is 5678
  ```
  - assign one pointer to another
  - make p1 point to where p2 points
  - p1 is assigned the same address as p2
- How about this one?
  ```
  *p1 = *p2;
  ```
  - assign the value pointed to by p1 to the value pointed to by p2
  - copy the content that p2 points to the content that p1 points

---

Assignment of Pointers (2/2)

```plaintext
p1
\rightarrow \square4
```

- `p1` points to the value `\square4`.
- `*p1` is the only number that can be directly assigned to pointer variables.

---

Initialize Pointer Variables

- C/C++ does not automatically initialize variables
- Initialize a pointer constant value 0 (a.k.a. null pointer)
  ```
  int* ptr = 0;
  ```
  - store the null pointer in ptr
  - ptr points to nothing
  - constant NULL is also equivalent
  ```
  EX: int* ptr = NULL;
  ```
- Constant 0 is the only number that can be directly assigned to pointer variables.

---

The new Operator

- Since pointers can refer to variables…
  - no real need to have a standard identifier
- Can dynamically allocate variables
  ```
  operator new creates variables
  ```
  - no identifiers to refer to them
  - just a pointer!
- Example: `p1 = new int;`
  ```
  - creates a new nameless variable, and assigns p1 to point to it
  - can access with *p1
  - use just like ordinary variable
Example of Pointer Manipulations (1/2)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int *p1, *p2;
    p1 = new int;
    *p1 = 45;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "p2 == " << p2 << endl;
    *p2 = 23;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
}
```

Example of Pointer Manipulations (2/2)

```cpp
p1 = new int(101);//initialize as well
*p2 = 77;
cout << "*p1 == " << *p1 << endl;
cout << "*p2 == " << *p2 << endl;
return 0;
```

Result on screen:

```
*p1 == 23
*p2 == 45
*p1 == 101
*p2 == 77
```

Points and Functions

- Pointers are full-fledged types
  - can be used just like other types
  - can be function parameters
  - can be returned from functions

- Example:
  ```cpp
  int* fFindPointer(double* p);
  ```
  - this declaration of function fFindPointer:
    - has one pointer-to-double parameter
    - returns pointer-to-int variable

Memory Management

- Heap (a.k.a. freestore)
  - reserved for dynamically-allocated variables
  - all new dynamic variables consume memory in freestore
  - if too many => could use all freestore memory

- Future new operations will fail if the freestore is full
Checking **new** Success (1/2)

- Older compilers:
  - test if null returned by call to **new**:
    ```
    int* p;
    p = new int;
    if (p == NULL)
    {
        cout << "Insufficient memory.\n";
        exit(1);
    }
    - if **new** succeeds, program continues
    ```

Checking **new** Success (2/2)

- For newer compilers, if new operation fails:
  - Program terminates automatically
  - Produces error message
  - Still good practice to use NULL check

Freestore Size

- Varies with implementations/machines
- Typically large
  - most programs won’t use all memory
  - but not for buggy programs or some particular applications
- Memory management
  - still good practice
  - solid software engineering principle
  - memory is always finite
  - regardless of how much there is!

The **delete** Operator

- De-allocate dynamic memory
  - when no longer needed
  - return memory to freestore
- Example:
  ```
  int* p;
  p = new int(5);
  .../some processing...
  delete p; //delete space that p points to
  - **de-allocate** dynamic memory pointed to by
  - pointer p
  - literally **destroys** memory space
### Dangling Pointers

- delete p;
  - destroy dynamic memory
  - but p still points the original address ⇒ called dangling pointer
  - if p is then dereferenced (*p)
    ⇒ unpredictable results! often disastrous!
- Avoid dangling pointers
  - assign pointer to NULL after delete:
    delete p;
    p = NULL;

### User-defined Pointer Types

- Define pointer types
  - able to declare pointers like other variables
  - eliminate need for "*' in pointer declaration
- Example: typedef int* tIntPtr;
  - define a new type alias
- Consider these declarations:
  tIntPtr p;
  int* p;
  - two forms are equivalent

### Pointer to Structure

- Access data in the struct through a variable by using "." operation
- C++ provides member access operator arrow (->) to retrieve components of user-defined variables
- Example: given a struct tPersonal_ID
tPersonal_ID myself = ("Charles",30);
tPersonal_ID* current;
current = &myself;
cout << current->name << endl;
cout << (*current).age << endl;

### Dynamic v.s. Automatic Variables

- Local variables
  - declared within function definition ⇒ Not dynamic
  - created when function is called
  - destroyed when function call completes
  - often called automatic variables ⇒ properties controlled for you
- Dynamic variables
  - created with new operator
  - created and destroyed while program runs
### Dynamic Arrays
- **Array variables**
  - actually pointer variables!
- **Standard array**
  - *fixed* dimensions for array
  - size for each dimension needs to be a constant.
- **Dynamic array**
  - size *not specified* at programming time
  - determined while program running

### Array Variables v.s. Pointers (1/3)
- Recall: arrays stored in memory addresses, sequentially
  - array variable "refers to" first indexed variable
  - so array variable is a kind of pointer variable!
- Example:
  ```c
  int a[10];
  int *p;
  // a and p are both pointer variables!
  ```

### Array Variables v.s. Pointers (2/3)
- Recall previous example:
  ```c
  int a[10];
typedef int* IntPtr;
IntPtr p;
  // a and p are pointer variables
  ```
- Performing assignments:
  ```c
  p = a; //LEGAL
  p now points where a points
  To first indexed variable of array a
  a = p; //ILLEGAL!
  //array pointer a is constant pointer
  ```

### Array Variables v.s. Pointers (3/3)
- **Array variable**
  ```c
  int a[10];
  //more than a pointer variable
  //"const int" type
  //array was allocated in memory already
  //variable a MUST point there...always! ⇒ cannot be changed!
  ```
- In contrast to ordinary pointers
  - which can (& typically do) change
Creating Dynamic Arrays

- Use new operator
  - dynamically allocate with pointer variable
  - treat like standard arrays

- Example:
  int iSize = 0;
  cin >> iSize;
  typedef double* DoublePtr;
  DoublePtr d;
  d = new double[iSize]; // size in brackets
  create dynamically allocated array variable d
  contain iSize elements of type double

Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - so should be destroyed at run-time
- Continue the previous example:
  ...
  delete [] d; // delete array that p points
  - de-allocate all memory for dynamic array
  - brackets [] indicate array is there
  - note that d still points there. ⇒ dangling!
  ⇒ should add "d = NULL," immediately

Pointer Arithmetic

- Can perform arithmetic on pointers
  - arithmetic over memory address
  - support addition/subtraction
  - compared using relational operators (==, !=, <, >, etc.)

- Example:
  int nums[100];
  int* iPtr;
  iPtr = &nums[0]; // equivalently, iPtr = nums;
  iPtr += 4;
  if (iPtr == &nums[4])
    cout << "jump to 4th element" << endl;
**Advanced Pointer Notation (1/2)**

- Access to *multi-dimensional* arrays can be made using pointer notation
- Example: Consider the declaration:
  ```cpp
  int nums[2][3] = {{16, 18, 20}, {25, 26, 27}};
  ```
  - create an array of elements and a set of pointer constants named `nums`, `nums[0]` and `nums[1]`
  - address of the first element in first row of `nums` is `nums[0]`
  - int variable pointed to by `nums[1]` is `nums[1][0]`

**Advanced Pointer Notation (2/2)**

<table>
<thead>
<tr>
<th>pointer notation</th>
<th>subscript notation</th>
<th>actual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>*nums[0]</td>
<td>nums[0][0]</td>
<td>16</td>
</tr>
<tr>
<td>*(nums[0]+1)</td>
<td>nums[0][1]</td>
<td>18</td>
</tr>
<tr>
<td>*(nums[0]+2)</td>
<td>nums[0][2]</td>
<td>20</td>
</tr>
<tr>
<td>*nums[1]</td>
<td>nums[1][0]</td>
<td>25</td>
</tr>
<tr>
<td>*(nums[1]+1)</td>
<td>nums[1][1]</td>
<td>26</td>
</tr>
<tr>
<td>*(nums[1]+2)</td>
<td>nums[1][2]</td>
<td>27</td>
</tr>
</tbody>
</table>

**Dynamic Multi-dimensional Arrays**

- Multi-dimensional arrays are *arrays of arrays*
  - various ways to create dynamic multi-dimensional arrays.
- Example:
  ```cpp
  typedef int* IntArrayPtr;
  IntArrayPtr* m = new IntArrayPtr[3];
  ```
  for (int idx = 0; idx < 3; idx++)
  ```cpp
  m[idx] = new int[4];
  ```
  - declare one array `m` of 3 IntArrayPtr pointers
  - make each allocated array of 4 integers
  - create one 3x4 dynamic array

**Alternative Array Manipulation**

- Use pointer arithmetic!
- *Step through* array without indexing(1):
  ```cpp
  for (int idx = 0; idx < arraySize; idx++)
  cout << *(d + idx) << " \n";
  ```
- Equivalent to:
  ```cpp
  for (int idx = 0; idx < arraySize; idx++)
  cout << d[idx] << " \n";
  ```
- Only addition/subtraction on pointers
  - no multiplication, division
- Can use ++ and -- on pointers
Example: 2-dimensional Dynamic Arrays

- Example:
  ```
  int *Mat1[4]; // 2nd dimension is fixed at
  for (int row = 0; row < 4; row++)
    Mat1[row] = new int[6]; // create 6 columns
  -4 rows Mat1[0], Mat1[1], Mat1[2] and
  Mat1[3] are declared
  - each row has 6 columns to be created
  Example: (most common)
  int **Mat2; // 2-level pointer
  Mat2 = new int *[4]; // create 4 rows dynamically
  for (int row = 0; row < 4; row++)
    Mat2[row] = new int [6]; // create 6 columns
  - both Mat2 and **Mat2 are pointers
  ```

Delete Dynamic Arrays

- After a dynamic array is of no use any more, deallocates the memory by delete operation
  - Clean reversely from last allocated memory
- Example: // reallocate a dynamic 5x9 matrix
  ```
  int ** Mat = new int *[5]; // create 5 rows
  for (int row = 0; row < 9; row++)
    Mat[row] = new int [9]; // create 9 columns
  // some processing
  for (int row = 0; row < 9; row++) // clean columns
    delete [] Mat[row];
  delete [] Mat; // clean rows
  Mat = NULL;
  ```

Shallow vs. Deep Copies

- Shallow copy (copy-by-address)
  - two or more pointers point to the same memory address
- Deep copy (copy-by-value)
  - two or more pointers have their own data
- Example:
  ```
  int *first, *second;
  first = new int[10];
  second = first; // shallow copy
  second = new int[10];
  for (int idx=0; idx<10; idx++) // deep copy
    second[idx] = first[idx];
  ```

Passing Arrays by Function

- When array is passed to a function, only pass the address of the first element
- Example: in main function
  ```
  int max = FindMax(array, size);
  ```
  in function declaration section
  ```
  int FindMax(int* val, int num)
  ```
  ```
  { ... }
  ```
  - Parametr receives the address of array array
  - val is one pointer
  - Another form:
    ```
    int FindMax(int val[] , int num)
    ```
### Returning an Array by Function (1/2)

- Array type pointers are NOT allowed as return-type of function.

- Example:
  ```c
  int [] someFunction(); //ILLEGAL!
  ```

- Instead return pointer to array base type:
  ```c
  int* someFunction();
  ```

- Return a `integer pointer` after function call.

- `in main (or caller)` function,

- `- Only ONE array (address) can be returned!`

### Returning an Array by Function (2/2)

- One more example:
  ```c
  int* display();
  void maint()
  {
    cout << *display() << endl;
  }

  int* display()
  {
    int* iptr = new int[0];
    int b[2] = {10, 20};
    for (int idx = 0; idx < 2; idx++)
      *iptr += b[idx];
    return iptr;
  }
  ```

### Allocating C-Strings

- A typical problem is to store an array of C-Strings.

- First declare an array of points to C-Strings

- Dynamically allocate space for C-Strings.

- Example:
  ```c
  char word[100]; char * wvec[50];
  ...
  while (cin >> word) {
    int len = strlen(word)+1;
    char * nword = new char [len];
    strcpy(nword, word);
    wvec[n] = nword;
    n++;
  }
  ```

### Expanding Dynamic Arrays

- A program can start with a small array and then expands it only if necessary.

- Example:
  ```c
  //assume MAX = 10 at first
  int * ivec = new int[MAX];
  ...
  //initialize n to 0
  while (cin >> ivec[idx]) {
    n++;
    if (n > MAX) {
      MAX *= 2;
      int * tmp = new int[MAX];
      for (int i = 0; i < n; i++)
        tmp[i] = ivec[i];
      delete [] ivec;
      ivec = tmp;
    }
  }
  ```
Common Programming Errors (1/2)

- Using a pointer to access nonexistent array elements
- Incorrectly apply address and indirect operators
  
  ```c
  int *ptr1 = &45;
  int *ptr2 = &(miles+10);
  ```
  - Illegal to take the address of a value
- Taking addresses of pointer constants
  
  ```c
  int nums[25];
  int * pt;
  pt = &nums;
  ```
  - Correct form: `pt = nums;`

Common Programming Errors (2/2)

- Taking addresses of a register variable
  - Internal registers do not have addresses
- Initializing pointer variables incorrectly
  
  ```c
  int *pt = 5;
  ```
  - `pt` is a pointer to an integer
  - Must be a valid address of another integer variable or NULL
- Forgetting to the bracket set, [], after the delete operator when dynamically deallocating memory

Summary

- Pointer is memory address
  - Provides indirect reference to variable
- Dynamic variables
  - Created and destroyed while program runs
- Freestore
  - Memory storage for dynamic variables
- Dynamically allocated arrays
  - Size determined as program runs