Pointers

Adapted from Dr. Raman Adaikkalavan
CS & Informatics
IUSB
Topics

• How are variables created?
• Basic Pointers
• New and Delete Operators
• Arrays and Pointers
• Multi Dimensional Arrays and Pointers
• Functions and Pointers
• Char Pointers
THE BASICS OF POINTERS
How is a variable created?

• `int i = 5;`

• Find 4 bytes in RAM
• Name that 4 bytes as `i`
  – CPU gets the starting address of `i`
  – Assigns that address to `i`
• Stores the value 5 in that 4 bytes: `(0..00101)`
int i = 3;
cout << i; //3
cout << &i; //8760 is the address of i

Variables and their addresses!
Pointers

```cpp
int i = 3;
cout << i; // 3
cout << &i; // 8760

int *p;
p = &i; // store address

cout << p; // 8760
cout << &p; // 9740
cout << *p; // 3
```

```
  i
  3

  p
  8760

8760  9740
```
int i = 3;
cout << i;   //3
cout << &i;  //8760

int *p;
p = &i;

cout << p   //8760
cout << &p   //9740
cout << *p   //3

int **q;    /* why */
q = &p;

cout << q   //9740
cout << &q   //5020
cout << *q   //8760
cout << **q  //3

q can change both p and i
"Pointing to" Example

• Consider:

```cpp
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;  // what is printed?
cout << *p1 << endl;
```
"Pointing to" Example

• Consider:
  
  ```
  v1 = 0;
  p1 = &v1;
  *p1 = 42;
  cout << v1 << endl; // what is printed?
  cout << *p1 << endl;
  ```

  Produces output:
  
  42
  42

  • *p1 and v1 refer to same variable
Pointer Assignments

• Pointer variables can be "assigned":
  ```c
  int *p1, *p2;
p2 = p1;
  ```
  – Assigns one pointer to another
  – "Make p2 point to where p1 points"

• Do not confuse with:
  ```c
  *p1 = *p2;
  ```
  – Assigns "value pointed to" by p1, to "value pointed to" by p2
Display 10.1  Uses of the Assignment Operator with Pointer Variables

\[
p1 = p2;
\]

Before:

\[
p1 \quad \quad \quad 8
\]

\[
p2 \quad \quad \quad 9
\]

\[
*p1 = *p2;
\]

Before:

\[
p1 \quad \quad \quad 8
\]

\[
p2 \quad \quad \quad 9
\]

After:

\[
p1 \quad \quad \quad 8
\]

\[
p2 \quad \quad \quad 9
\]
Exercise – What each one is doing?

datatype *identifier;

int i=5;
int* p1;
int * p2;
int *p3;
int *p4, p5;
int *p6, *p7;
• int i = 5;
• int* p1;
• int * p2;
• int *p3;
• int *p4, p5;
• int *p6, *p7;
//p1 = i;  // error
p1 = &i;
cout << p1 << endl;
cout << &i << endl;
p2 = p1;
cout << p2 << endl;
p3 = p2;
cout << *p3 << endl;
NEW AND DELETE OPERATORS

STATIC VS. DYNAMIC MEMORY!!
Static vs. Dynamic Memory!

- **Static integer variable:**
  - `int x = 5;`

- **Dynamic integer variable:**
  - `int *p;` // pointer to a variable
  - `p = new int(5);` // Dynamically allocate an integer
The new Operator

• Since pointers can refer to variables...
  – No "real" need to have a standard identifier/variable
• Can dynamically allocate variables
  – Operator **new** creates **nameless** variables
    • No identifier names to refer to them
    • Just a pointer!
• int *p1;
• p1 = new int (10);
  – Creates new "**nameless**" variable, assigns p1 to "point to" it, and stores the value 10
  – Can access the value with *p1
    • Use just like an ordinary variable
Display 10.2  Basic Pointer Manipulations

1    //Program to demonstrate pointers and dynamic variables.
2    #include <iostream>
3    using std::cout;
4    using std::endl;

5    int main()
6    {
7        int *p1, *p2;
8        p1 = new int;
9        *p1 = 42;
10       p2 = p1;
11       cout << "*p1 == " << *p1 << endl;
12       cout << "*p2 == " << *p2 << endl;
13       *p2 = 53;
14       cout << "*p1 == " << *p1 << endl;
15       cout << "*p2 == " << *p2 << endl;
16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;
20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }

**Sample Dialogue**

*p1 == 42
*p2 == 42
*p1 == 53
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53

Hope you got the point of this example!
Display 10.3  Explanation of Display 10.2

(a)  
int *p1, *p2;

(b)  
p1 = new int;

(c)  
*p1 = 42;

(d)  
p2 = p1;

(e)  
*p2 = 53;

(f)  
p1 = new int;

(g)  
*p1 = 88;
New Operator (Contd.)

• New dynamic variable of specified type

```cpp
int *p;
p = new int(1000);
```

```cpp
myType *p;
p = new myType(123.4, 70);
```
Dynamic, Local, and Global Variables

- **Dynamic variables**
  - Created with `new` operator
  - Created and destroyed @ run time i.e., while program runs

- **Local variables** (sometimes automatic variables)
  - Declared within function definitions
    - `main()`, `print()`, ...
  - **Not** dynamic
    - Created when a function is called
    - Destroyed when the function exits

- **Global variables**
  - Outside any function or class, including outside `main()`
  - Sometimes called *statically allocated variables*
  - Created @ compile time and destroyed when program stops
Basic Memory Management

• Special Memory Area: Heap
  – a.k.a. "freestore"
  – Reserved for dynamically-allocated variables
  – All new dynamic variables consume memory in freestore
    • If too many new → could use all freestore memory
      – Old compilers return NULL (nothing by integer constant 0)
      – Standard compilers terminate the program
      – If a leak occurs, can crash your system!
  • Future "new" operations will fail if freestore is "full"
delete Operator

- **De-allocate** dynamic memory
  - When the dynamic memory is no longer needed by the program
  - Returns memory created by `new` back to freestore
  - Example:
    ```
    int *p;
    p = new int(5);
    if(p == NULL) // check to see if allocation was successful
      {cout << "Error"; exit(1);}
    //Some processing using p ...
    delete p; // p is no longer needed
    ```
  - De-allocates dynamic memory "pointed to by pointer p"
    - Return memory to heap!
What is the output?

1. `int *p = new int (10), *x;`
2. `x = p; //pointer assignment`
3. `cout << *p;`
4. `delete p;`
5. `cout << *p`
6. `cout << *x;`
Dangling Pointers

1. int *p = new int (10), *x;
2. x = p;
3. cout << *p;
4. delete p;
5. cout << *p
6. cout << *x;

– In step 3, it prints 10.
– In step 4, p is deleted. So p becomes undefined, as you do not know where it is pointing to.
– Thus, result of step 5 is undefined. CAUTION!!
– Result of step 6 is also undefined, dangling pointer
What is the output?

1. int *p, v = 10;
2. p = &v;
3. cout « *p;
4. delete p;
5. cout « *p;
6. cout « v;
What is the output?

1. `int *p, v = 10;`
2. `p = &v;`
3. `cout << *p;`
4. `delete p;`
5. `cout << *p;`
6. `cout << v;`

Step 4: Program will Crash! Blue Screen Of Death!
You can only use `delete` when new operator was used
New

• When new used, it creates a Nameless Location
ARRAYS AND POINTERS
Arrays

int ia[6];

- Allocates consecutive spaces for 6 integers

- How much space is allocated?
int ia[6];
• Allocates consecutive spaces for 6 integers
• How much space is allocated?
  \[ 6 \times \text{sizeof(int)} \]
• Also creates \( \text{ia} \) which is effectively a \textit{constant pointer} to the first of the six integers \textit{e.g., value @ location 4200 CANNOT be changed; it is the base address of the array;}
• What does \( \text{ia}[4] \) mean?
int ia[6];

- Allocates consecutive spaces for 6 integers
- How much space is allocated?
  \[ 6 \times \text{sizeof(int)} \]
- Also creates \( \text{ia} \) which is effectively a constant pointer to the first of the six integers
- **What does \( \text{ia}[4] \) mean?**
- **Multiply 4 by \text{sizeof(int)}. Add to \text{ia} and dereference yielding:**

\[ \text{ia}[4] \rightarrow 8000 + 16 \]
int ia[6];

- **Note:** \( ia \equiv \&ia[0] \)
- **Never say,** “*Pointer and arrays are exactly the same thing!!!***”

```c
int *ip;
ip = ia; /* Okay */
ia = ip; /* Illegal, value @ location 4200 cannot be changed */
```
Arrays

int ia[6];

• **Note:** \( ia \equiv \&ia[0] \)

• Never say, “**Pointer and arrays are exactly the same thing!!!**”
  
  ```
  int *ip;
  ip = new int (50);
  ia = ip; /* Illegal, value @ location 4200 cannot be changed */
  ```

![Diagram showing memory allocation and pointer values](image)
Arrays

- $ia[4]$ means $*(ia + 4)$
- which means $*(ia + 4 * \text{sizeof(int)}))$
- Or $*(ia + 4 * \text{sizeof(*ia))}$
Dynamic Arrays

- `int ia[6]; // cannot change the ia pointer (constant)`

- `int *ip = new int[6]; // ip pointer can be changed`

```c
ip = &ia;
```
Array Of Pointers

int v1=5, v2=6, v3=4, v4=9;
int *arr1[4]; //array of 4 integer pointers i.e.,
it can point to 4 variables, 4 arrays, or a combination

arr1[0] = &v1;
arr1[1] = &v2;
arr1[2] = &v3;
arr1[3] = &v4;
MULTIDIMENSIONAL ARRAYS AND POINTERS
Array of Pointers
Multi Dimensional Arrays

double * emp[3] = {}; //store 3 information about one employee

double * empId = new double(4); //store id
double empHours = 42; //store hours
double *empSal = new double [2]; //array to store straight pay & overtime pay

*(empSal) = 1500; //store straight time pay

*(empSal+1) = 200; //store overtime pay
Multidimensional arrays

\[
\text{int twoDPtr}[3][4];
\]

- **Type**: \[\text{int}\]
- **Address**: \[\text{twoDPtr}[3][4]\]
- **Number of Rows**: 3
- **Number of Columns**: 4

Declaration at compile time, i.e. size must be known
How does a two dimensional array work?

Row Major Order

0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3
Advantage

- Using Row Major Order allows visualization as an array of arrays

\[
\text{twoDPtr}[1]
\]

\[
\begin{array}{cccc}
0,0 & 0,1 & 0,2 & 0,3 \\
\end{array}
\]

\[
\begin{array}{cccc}
1,0 & 1,1 & 1,2 & 1,3 \\
\end{array}
\]

\[
\begin{array}{cccc}
2,0 & 2,1 & 2,2 & 2,3 \\
\end{array}
\]

\[
\text{twoDPtr}[1][2]
\]

\[
\begin{array}{cccc}
0,0 & 0,1 & 0,2 & 0,3 \\
\end{array}
\]

\[
\begin{array}{cccc}
1,0 & 1,1 & 1,2 & 1,3 \\
\end{array}
\]

\[
\begin{array}{cccc}
2,0 & 2,1 & 2,2 & 2,3 \\
\end{array}
\]
Arrays and addresses

- One dimensional array
  \[
  \text{int } \text{ia}[6];
  \]
  \[
  \text{ia } \equiv \&\text{ia}[0]
  \]

- Address of beginning of array:
  \[
  \text{ia } \equiv \&\text{ia}[0]
  \]

- Two dimensional array
  \[
  \text{int } \text{ia}[3][6];
  \]
  \[
  \text{ia } \equiv \&\text{ia}[0]
  \]
  \[
  \text{ia } \equiv \&\text{ia}[0][0]
  \]

  \[
  \text{Address of row 1:}
  \]
  \[
  \text{ia}[1] \equiv \&\text{ia}[1][0]
  \]

  \[
  \text{Address of row 2:}
  \]
  \[
  \text{ia}[2] \equiv \&\text{ia}[2][0]
  \]
Element access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:
  \[
  \text{row\_index} * \text{sizeof}(\text{row})
  \]
  \[
  \text{row\_index} * \text{Number\_of\_columns} * \text{sizeof(arr\_type)}
  \]
  This plus \textit{address of array} gives address of first element of desired row
- Add \text{column\_index} * \text{sizeof(arr\_type)} to get actual desired element
**(name + innermost element) + nextlevel) + nextlevel)+..**

<table>
<thead>
<tr>
<th>twoDPtr</th>
<th>*(twoDPtr+0)</th>
<th>*(twoDPtr+0)+0</th>
<th>*(twoDPtr+0)+1</th>
<th>*(twoDPtr+0)+1+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>twoDPtr[0]</td>
<td>twoDPtr[0][0]</td>
<td>twoDPtr[0][1]</td>
<td>twoDPtr[0][2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>twoDPtr+1</th>
<th>*(twoDPtr+1)</th>
<th>*(twoDPtr+1)+0</th>
<th>*(twoDPtr+1)+1</th>
<th>*(twoDPtr+1)+2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>twoDPtr[1]</td>
<td>twoDPtr[1][0]</td>
<td>twoDPtr[1][1]</td>
<td>twoDPtr[1][2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>twoDPtr+2</th>
<th>*(twoDPtr+2)</th>
<th>*(twoDPtr+2)+0</th>
<th>*(twoDPtr+2)+1</th>
<th>*(twoDPtr+2)+2</th>
</tr>
</thead>
</table>
First dimension

0
1
2
Second Dimension
\[ \ast(\ast(\ast((\text{name} + \text{innermost element}) + \text{nextlevel}) + \text{nextlevel}) + \ldots) \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>eL</td>
<td>eL+1</td>
<td>eL+1</td>
</tr>
<tr>
<td>1</td>
<td>eL</td>
<td>eL+1</td>
<td>eL+1</td>
</tr>
<tr>
<td>2</td>
<td>eL</td>
<td>eL+1</td>
<td>eL+1</td>
</tr>
</tbody>
</table>

*eL or *(eL+0)

**eL or *(eL+0)+0

Same address

int eL[3][3] or int **eL