Pointers

Table of Contents

- Dynamic Variables
- Pointer Declaration
- Pointer References
- Pointer Manipulation
- Addressing: Direct & Indirect
- Record Pointers
- Arrays of Pointers
- Pointer Expressions
- Dynamic Storage
- Dynamic Memory Problems
- Reference Variables
Dynamic Variables

✿ Static Variables
– Size is fixed throughout execution
– Size is known at compile time
– Space/memory is allocated at compilation

✿ Dynamic Variables
– Created during execution
  † "dynamic allocation"
– No space allocated at compilation time
– Size may vary
  † Structures are created and destroyed during execution.
– Knowledge of structure size not needed
– Memory is not wasted by non-used allocated space.
– Storage is required for addresses.

✿ Example of Pointers
– Assume:
  Houses represent data
  Addresses represent the locations of the houses.
– Notice:
  To get to a house you must have an address.
  No houses can exist without addresses.
  An address can exist without a house (vacant lot / NULL pointer)
**Pointer Declaration**

**Pointer Type**
- Simple type of variables for storing the memory addresses of other memory locations

**Pointer Variables Declarations**
- The asterisk ‘*’ character is used for pointer variable declarations:

  ```
  int* iptr;  // recommended form
  float* fptr;
  ```

  - `iptr` is a pointer to an integer
  - `fptr` is a pointer to a real

  ```
  int* iptr1, iptr2;
  ```

  - Given the declaration:

  ```
  int* iptr1;
  int iptr2;
  ```

  † Declares `iptr1` to be a pointer variable, but `iptr2` is a simple integer variable.

  ```
  typedef int* intptr;
  intptr iptr1;
  ```

  † Declare all pointer variables in separate declaration statements.

  - Pointer Type Definitions:

  ```
  strong type declaration (preferred)
  ```
**Address Operator:** &
- Unary operator that returns the hardware memory location address of its operand.

Given:

```c
int* iptr1;
int* iptr2;
int numa, numb;
numa = 1;
numb = 2;
```

- **Address Assignment:**

```c
iptr1 = &numa;
iptr2 = &numb;
```

**Dereference / Indirection Operator:** *
- Unary ‘pointer’ operator that returns the memory contents at the address contained in the pointer variable.

- **Pointer Output:**

```c
cout << iptr1 << *iptr1 << endl;
cout << iptr2 << *iptr2 << endl;
```

- **Results:**

```
0xF4240 1
0x3B9ACA00 2
```
NULL Pointer
- Pointer constant, address 0
- Named constant in the `<stddef.h>` include header
- Represents the empty pointer
  † points nowhere, unique pointer/address value
- Symbolic/graphic representations: ❧ ✿
- Illegal: *NULL
### Pointer Diagrams

- **Given (text/code representation) Graphic representation**

```c
#include <stddef.h>
void main()
{
    int* iptr1 = NULL;
    int* iptr2 = NULL;
    int numa, numb;
    numa = 1;
    numb = 2;
}
```

<table>
<thead>
<tr>
<th>Graphic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram 1" /></td>
<td><code>iptr1 = &amp;numa;</code>&lt;br&gt;<code>iptr2 = &amp;numb;</code></td>
</tr>
<tr>
<td><img src="image2.png" alt="Diagram 2" /></td>
<td><code>*iptr2 = *iptr1 - 1;</code>&lt;br&gt;<code>iptr2 = iptr1;</code></td>
</tr>
<tr>
<td><img src="image3.png" alt="Diagram 3" /></td>
<td><code>iptr2 = *iptr1;</code>&lt;br&gt;<code>*iptr1 = iptr2;</code></td>
</tr>
</tbody>
</table>

### Pointer Assignments

- Independent from initial code above.

- #1

- #2

- #3
**Addressing: Direct & Indirect**

**Direct Addressing**
- normal variable access
- non-pointer variables represent one-level of addressing
- non-pointer variables are addresses to memory locations containing data values.
- compilers store variable information in a “symbol table”:

<table>
<thead>
<tr>
<th>symbol</th>
<th>type</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int</td>
<td>0xF4240</td>
</tr>
<tr>
<td>iptr</td>
<td>pointer (int)</td>
<td>0xF4241</td>
</tr>
</tbody>
</table>

- compilers replace non-pointer variables with their addresses & fetch/store operations during code generation.

**Indirect Addressing**
- accessing a memory location’s contents thru a pointer
- pointer variables represent two-levels of addressing
- pointer variables are addresses to memory locations containing addresses.
- compilers replace pointer variables with their addresses & double fetch/store operations during code generation.

**Note: indirect addressing required to dereference pointer variable.**

```
x     = 28;
iptr  = &x;
```

**MEMORY**

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF4239</td>
</tr>
<tr>
<td>0xF4240</td>
</tr>
<tr>
<td>0xF4241</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>0xF4240</td>
</tr>
<tr>
<td>???</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Record Pointers

Pointers to structures:
- Given:

```c
const int f3size = 20;
struct rectype {
    int field1;
    float field2;
    char field3[f3size];
};
typedef rectype *recPtr;
rectype rec1(1, 3.1415,"pi");
recPtr r1ptr;
r1ptr = &rec1;
```

Member Access
- Field Access Examples:

```c
#include <iostream>

int main() {
    cout << (*r1ptr).field1
        << (*r1ptr).field2
        << (*r1ptr).field3;
    return 0;
}
```

- Errors:

```c
#include <iostream>

int main() {
    cout << *r1ptr.field1
        << *r1ptr.field2
        << *r1ptr.field3;
    return 0;
}
```

Arrow Operator
- Short-hand notation:

```c
#include <iostream>

int main() {
    cout << r1ptr->field1
        << r1ptr->field2
        << r1ptr->field3;
    return 0;
}
```
Arrays of Pointers

 Declarations:

- Given:

```
const int size = 20;
struct rectype {
    int field1;
    float field2;
    char field3[size];
};
typedef rectype *recPtr;
rectype rec1(1, 3.1415, "pi");
recPtr rayPtrs[size];
rayPtrs[size-1] = &rec1;
```

 Member Access

- Field Access Examples:

```
cout << (*rayPtrs[size-1]).field1
    << (*rayPtrs[size-1]).field2
    << (*rayPtrs[size-1]).field3 ;
```

 Arrow Operator

- Short-hand notation:

```
cout << rayPtrs[size-1]->field1
    << rayPtrs[size-1]->field2
    << rayPtrs[size-1]->field3 ;
```
Arrays == Pointers
- Non-indexed Array variables are considered pointers in C
- Array names as pointers contain the address of the zero element (termed the base address of the array).

Given:

```c
const int size = 20;
char name[size];
char *person ;
person = name ;
person = &name[0] ;
```

Pointer Indexing
- All pointers can be indexed,
  (logically meaningful only if the pointer references an array).
- Example:

  ```c
  person[0] = ' ';
  person[size-2] = '.';
  ```

Logical Expressions
- NULL tests:
  ```c
  if (!person) //true if (person == NULL)
  ```

- Equivalence Tests:
  ```c
  if (person == name)
  //true if pointers reference
  //the same memory address
  ```
Heap (Free Store, Free Memory)
- Area of memory reserved by the compiler for allocating & deallocating to a program during execution.

Operations:

<table>
<thead>
<tr>
<th>C++</th>
<th>function</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>new type</td>
<td>allocation</td>
<td>malloc(# bytes)</td>
</tr>
<tr>
<td>delete pointer</td>
<td>deallocation</td>
<td>free pointer</td>
</tr>
</tbody>
</table>

NULL is returned if the heap is empty.

Allocation

```c
char* name;
int* iptr;
// C++
name = new char;
iptr = new int[20];
// initialization
name = new char (‘A’);
```

pointer typecasts required
dynamic array allocation

Deallocation

```c
// C++
delete name;
delete [] iptr;
//delete [20] iptr;
```

free(name);

Pointers are undefined after deallocation.
Dynamic Memory Problems

Given:

```c
typedef int *intPtr;
intPtr iptr1, iptr2;
```

Garbage

- Previously allocated memory that is inaccessible thru any program pointers or structures.
- Example:

  ```
  iptr1 = new int (6);
  iptr1 = NULL;
  ```

Aliases

- Two or more pointers referencing the same memory location.
- Example:

  ```
  iptr1 = new int (6);
  iptr2 = iptr1;
  ```

Dangling References

- Pointers that reference memory locations previously deallocated.
- Example:

  ```
  iptr1 = new int (6);
  iptr2 = iptr1;
  delete iptr1;
  ```

memory leaks
Reference Variables

Reference Variable Declarations
- The and ‘&’ character is used for reference variable declarations:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Reference Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int&amp; iptr;</td>
</tr>
<tr>
<td>float</td>
<td>float &amp;fptr1, &amp;fptr2;</td>
</tr>
</tbody>
</table>

Reference variables are aliases for variables.

Pointer Differences
- Reference variables do NOT use the address and dereference operators (& *).
- Compiler dereferences reference variables transparently.
- Reference variables are constant addresses, assignment can only occur as initialization or as parameter passing, reassignment is NOT allowed.
- Examples:

```
char achar = ‘A’;
char& chref = achar;
//char* chptr = &achar;
chref = ‘B’;
//achar = ‘B’;
//*chptr = ‘B’;
```

Purpose
- Frees programmers from explicitly dereferencing accessing, (in the same way nonpointer variables do).
- ‘Cleans up the syntax’ for standard C arguments and parameters.