Pointers

- Table of Contents

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;
float    *fptr;
```

recommended form

common declaration

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

  - iptr is a pointer to an integer
  - fptr is a pointer to a real
  - Given the declaration:

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

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

  - Equivalent declaration:

```
typedef  int *intPtr;
intPtr   iptr1;
```

† Declare all pointer variables in separate declaration statements.

  - Pointer Type Definitions:

strong type declaration (preferred)

Pointer References

- Address Operator: **&**
  - Unary operator that returns the hardware memory location address of it's operand.

Given:

```
int*     iptr1;
int*     iptr2;
int      numa, numb;

numa = 1;
numb = 2;
```

  - Address Assignment:

```
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:

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

  - Results:

```
0xF4240          1
0x3B9ACA00       2
```

Pointer References

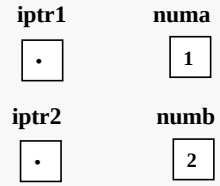
- NULL Pointer
  - Pointer constant, address 0
  - Named constant in the <stddef.h> include header
  - Represents the empty pointer
    - † points nowhere , unique pointer/address value
  - Symbolic/graphic representations:  
  - Illegal: ***NULL**

Pointer Manipulation

- Pointer Diagrams
  - Given (text/code representation) Graphic representation

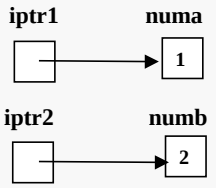
```
#include <stddef.h>
void main()
{
int*     iptr1 = NULL;
int*     iptr2 = NULL;
int      numa, numb;

numa = 1;
numb = 2;
}
```



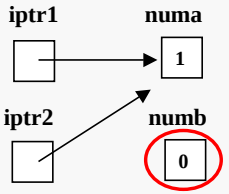
- Pointer Assignments
  - Independent from initial code above.
  - #1

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



  - #2

```
*iptr2 = *iptr1 - 1;
iptr2  = iptr1;
```



  - #3

```
iptr2  = *iptr1;
*iptr1 = iptr2;
```

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":

symbol	type	. . .	address
x	int	…	0xF4240
iptr	pointer (int)	…	0xF4241

  - 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	
address	*contents*
. . .	. . .
0xF4239	???
0xF4240	28
0xF4241	0xF4240
0xF4241	???
. . .	. . .

Record Pointers

- Pointers to structures:
  - Given:

```
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:

```
cout   << (*r1ptr).field1
       << (*r1ptr).field2
       << (*r1ptr).field3 ;
```

Note: parenthesis required due to operator precedence; without compiler attempts to dereference fields.

  - Errors:

```
cout        << *r1ptr.field1
            << *r1ptr.field2
            << *r1ptr.field3 ;
```

- Arrow Operator
  - Short-hand notation:

```
cout   << r1ptr->field1
       << r1ptr->field2
       << r1ptr->field3 ;
```

Note: -> is an ANSI "C" pointer member selection operator. Equivalent to:

(*pointer).member

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 ;
```


Pointer Expressions

- 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:

```
const int      size = 20;

char           name[size];
char           *person ;

person = name ;
person = &name[0] ;
```

equivalent assignments

Does not create a copy, (no memory allocation)

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

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

- Logical Expressions
  - NULL tests:

preferred check

```
if (!person) //true if (person == NULL)
```

  - Equivalence Tests:

```
if (person == name)
//true if pointers reference
//the same memory address
```

pointer types must be identical

Dynamic Storage

- Heap (Free Store, Free Memory)
  - Area of memory reserved by the compiler for allocating & deallocating to a program during execution.
- Operations:

C++	function	C
new type	allocation	malloc(# bytes)
delete pointer	deallocation	free pointer

NULL is returned if the heap is empty.

- Allocation

pointer typecasts required

```
char*          name;
int*           iptr ;
//  C++                          C
name = new char ;        name = (char *) malloc(sizeof(char));
iptr = new int [20] ;    iptr = (char *) malloc(20 * sizeof(char));
//initialization
name = new char ('A') ;
```

dynamic array allocation

- Deallocation

```
//  C++                          C
delete   name ;          free(name);
delete [] iptr ;
//delete [20] iptr ;
```

Pointers are undefined after deallocation.

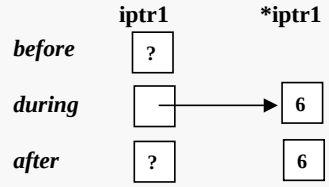
Dynamic Memory Problems

- Given:

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

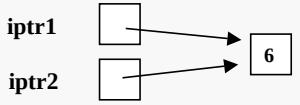
- Garbage
  - Previously allocated memory that is inaccessible thru any program pointers or structures.
  - Example:

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



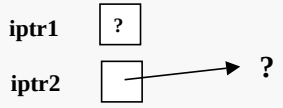
- Aliases
  - Two or more pointers referencing the same memory location.
  - Example:

```
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:

```
int&        iptr;
float       &fptr1, &fptr2;
```

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.