Pointers

3. Pointers

1. Table of Contents
2. Dynamic Variables
3. Memory and Addresses
4. Pointer Declaration
5. Pointer References
6. Pointer References (cont)
7. Pointer Manipulation
8. Pointers and Type
9. Addressing: Direct & Indirect
10. Record Pointers
11. Pointer Expressions
12. Dynamic Storage
13. Allocating Arrays
14. Allocating Arrays (cont)
15. Resizing an Array
16. Allocation Failure
17. Allocation Failure (cont)
18. Reference Pointer Parameters
19. Value Pointer Parameters
20. const Pointer Parameters
21. const Pointers
22. const Summary
23. Pointer Array Arithmetic
24. Incrementing Pointers
25. Array of Structs Pointer
26. Pointer Incrementing Abuse
27. Arrays of Pointers
28. Dynamic Memory Problems
29. Reference Variables

Dynamic Variables

2. 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)
3. Pointers

Memory and Addresses

On modern computers, memory is organized in a manner similar to a one-dimensional array:

- Memory is a sequence of bytes (8 bits)
- Each byte is assigned a numerical address, similar to array indexing
- Addresses are nonnegative integers; valid range is determined by physical system and OS memory management scheme
- OS (should) keep track of which addresses each process (executing program) is allowed to access, and attempts to access addresses that are not allocated to a process should result in intervention by the OS
- OS usually reserves a block of memory starting at address 0 for its own use
- Addresses are usually expressed in hexadecimal (base 16), typically indicated by use of a prefix: 0xF4240

Memory Organization

- Run-time stack used for statically allocated storage
- Heap used for dynamically allocated storage

3. Pointers

Ptr 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;
  ```

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

- Given the equivalent declaration:

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

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

- Typedef declaration:

  ```
  typedef int* intPtr;
  ```

  ```
  intPtr iptr1;
  ```

  - Declare all pointer variables in separate declaration statements.

- Pointer Type Definitions:

  ```
  int* iptr1;
  ```

  ```
  int iptr2;
  ```

  - Strong type declaration (preferred)
### Pointer References

#### 3. Pointers

**Address Operator: & (ampersand)**

Unary operator that returns the hardware memory location address of its operand.

**Address Assignment:**

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

**Dereference / Indirection Operator:** *

(asterisk)

Unary ‘pointer’ operator that returns the memory contents at the address contained in the pointer variable.

**Pointer Output:**

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

(Possible) results:

| 0xF4240  | 1 |
| 0x3B9ACA00 | 2 |

**NULL Pointer**

- Pointer constant, address 0
- Named constant in the `<cstdlib>` include header (`<stddef.h>` old style header).
- Represents the empty pointer
  ➔ points nowhere, unique pointer/address value
- Illegal: NEVER dereference a pointer that equals NULL

![Error Message]

The instruction at "0x04B646" referenced memory at "0x00000001". The memory could not be "read".

- Click on OK to terminate the program.
- Click on OK to debug the program.
3. Pointers

**Pointers and Type**

Pointers have type:
- the type of a pointer is determined by the type of target that is specified in the pointer declaration.

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

- here, `iptr1` and `iptr2` are pointers to `int` (type `int*`).
- it is a compile-time error to assign a non-pointer value to a pointer:

```c
*iptr2 = *iptr1; // error: assign int to int*
```

or vice versa:

```c
*iptr1 = iptr2; // error: assign int* to int
```

**Typecasts and pointers:**
- the assignments above would be legal if an explicit typecast were used:

```c
iptr2 = (int*) *iptr1; // legal
```

```c
typedef int* iPtr;
```

```c
#define iPtr (*iptr1); // legal
```

```c
*iptr1 = int(iptr2); // legal
```

**However,** be very cautious with this sort of code. It rarely, if ever, makes much sense to assign a pointer a value that's not either another pointer, or obtained by using the dereference operator.
3. Pointers

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>int*</td>
<td>0xF4244</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.

Record Pointers

Pointers to structures:
- Given:

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

Member Access
- Field Access Examples:

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

Note: parentheses are required due to operator precedence; without () compiler attempts to dereference fields.

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

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

Note: -> is an ANSI “C” pointer member selection operator. Equivalent to: (*pointer).member

Arrow Operator
- Short-hand notation:

```c
out << r1ptr->field1
 << r1ptr->field2
 << r1ptr->field3;
```
3. Pointers

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];
```

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

Example:

```
person[0] = ' ';  //true if pointers reference the same memory address
person[size-2] = '.';
```

Logical Expressions
NULL tests:

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

Equivalence Tests:

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

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
```

With most compilers, `name` is returned if the heap is empty. However, see slide 3.16 for a caveat ...

Allocation

```
char* name;
int* iptr;
```

```
//C++ // C
name = new(nothrow)char;
name = (char *) malloc(sizeof(char));
```

```
iptr = new(nothrow) int [20];
```

```
//initialization
name = new char ('A');
iptr = 
```

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

```
//delete [20] iptr;
```

```
//dynamic array allocation
```

Deallocation

```
//C++ // C
delete name;
name = NULL;
```

```
delete [] iptr;
iptr = NULL;
```

```
delete [20] iptr;
```

```
//free(iptr);
```

Pointers are undefined after deallocation and should be set to NULL.
3. Pointers

Declaration Syntax

```c
int Size;
cin >> Size; // dynamic value
// use as array dim
char* Name = new (nothrow) char[Size];
int* Scores;
Scores = new (nothrow) int[Size];
Size = 4 * Size + 1; // does NOT change array
```

Effect of array allocation via `new`

- **Scores**: 3F42740
- **Address returned by new; value of Scores**: 3F42740, 3F42744, 3F42748, 3F42750
- **Storage space is allocated contiguously in memory**

Use like any statically-allocated array

```c
strcpy(Name, "Fred G Flintstone"); // static size
```

```
for (int Idx = 0; Idx < Size; Size++)
    Scores[Idx] = 0;
SortScores(Scores, Size);
```

Deallocation

```c
delete [] Name;
delete [] Scores;
delete [20] Scores; // including dim is optional
```

Failure to explicitly `delete` a dynamic variable will result in that memory **NOT** being returned to the system, even if the pointer to it goes out of scope.

This is called a “**memory leak**” and is evidence of poor program implementation.

If large dynamic structures are used (or lots of little ones), a memory leak can result in depletion of available memory.

```c
// WARNING
delete Name;
// May not release array memory, undefined results
```
### Resizing an Array

Growing a dynamically-allocated array

```c
int* newArray = new int[newSize];

// copy contents of old array into new one
for (int Idx = 0; Idx < oldCapacity; Idx++)
    newArray[Idx] = Scores[Idx];

// delete old array
delete [] Scores;

// retarget old array pointer to new array
Scores = newArray;

// clean up alias
newArray = NULL;
```

### Allocation Failure

An invocation of operator `new` will fail if the heap does not contain enough free memory to grant the request.

Traditionally, the value `NULL` has been returned in that situation. However, the C++ Standard changes the required behavior. By the Standard, when an invocation of `new` fails, the value returned may or may not be `NULL`; what is required is that an `exception` be thrown. We do not cover catching and responding to exceptions in this course.

Fortunately, for the present, most C++ language implementations will continue to guarantee that `NULL` is returned in this case.

Better still, the Standard provides a way to force a `NULL` return instead of an exception throw:

```c
const int Size = 20;
int* myList = new(nothrow) int[Size];
```

Use of this syntax will guarantee that `myList` will be assigned `NULL` if the allocation fails.

```c
#pragma warning (disable:4291)
```

// to turn off noexcept warning

The following program attempts to allocate an array, initialize it, and then display its contents. However, the allocation will almost certainly fail.

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int Count;
    int* t;
    const int Size = 900000000;
    int* myList = new(nothrow) int[Size];
    if (myList == NULL) {
        cout << "Allocation failed!!" << endl;
        return( EXIT_FAILURE );
    }
    for (t = myList, Count = 0; Count < Size; Count++, t++)
        *t = Count;
    for (t = myList, Count = 0; Count < Size; Count++, t++)
        cout << t << setw(5) << *t << endl;
    return( EXIT_SUCCESS );
}
```

The following program attempts to allocate an array, initialize it, and then display its contents. However, the allocation will almost certainly fail.

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int Count;
    int* t;
    const int Size = 900000000;
    int* myList = new(nothrow) int[Size];
    if (myList == NULL) {
        cout << "Allocation failed!!" << endl;
        return( EXIT_FAILURE );
    }
    for (t = myList, Count = 0; Count < Size; Count++, t++)
        *t = Count;
    for (t = myList, Count = 0; Count < Size; Count++, t++)
        cout << t << setw(5) << *t << endl;
    return( EXIT_SUCCESS );
}
```

In C++, all function parameters are, by default, passed by value. When passing a pointer as a parameter to a function, you must decide how to pass the pointer.

If the called function needs to modify the value of the pointer, you must pass the pointer by reference:

```cpp
void growArray(int*& Array, const int oldSize, const int newSize) {
    assert(newSize > oldSize);
    int* tempArray = new int[newSize];
    Copy(tempArray, Array, oldSize);
    delete[] Array;
    Array = tempArray; // modifies VALUE of Array
    tempArray = NULL; // is this statement necessary?
}
```

In C++, all function parameters are, by default, passed by value. When passing a pointer as a parameter to a function, you must decide how to pass the pointer.

If the called function needs to modify the value of the pointer, you must pass the pointer by reference:

```cpp
void growArray(int*& Array, const int oldSize, const int newSize) {
    assert(newSize > oldSize);
    int* tempArray = new int[newSize];
    Copy(tempArray, Array, oldSize);
    delete[] Array;
    Array = tempArray; // modifies VALUE of Array
    tempArray = NULL; // is this statement necessary?
}
```
3. Pointers

Value Pointer Parameters

If the called function only needs to modify the value of the target of the pointer, you may pass the pointer by value:

```c
void Copy(int* Target, int* Source, const int Dim) {
    for (int Idx = 0; Idx < Dim; Idx++)
        Target[Idx] = Source[Idx];
}
```

Copy() copies the target of one pointer to the target of another pointer. Neither pointer is altered.

This is termed a side-effect. Considered poor practice. Better to pass pointers by reference to indicate the change of target, (or better still to explicitly pass the pointer by const but not the target).

```c
void Copy(int* const Target,
          const int* const Source,
          const int Dim);
```

`const` Pointer Parameters

Passing a pointer by value is somewhat dangerous. As shown in the implementation of Copy() on the previous slide, if you pass a pointer to a function by value, the function does have the ability to modify the value of the target of the pointer. (The called function receives a local copy of the pointer’s value.)

This is objectionable if the function has no need to modify the target. The question is: how can we pass a pointer to a function and restrict the function from modifying the target of that pointer?

```c
void Print(const int* Array, const int Size) {
    for (int Idx = 0; Idx < Size; Idx++) {
        cout << setw(5) << Idx << setw(8) << Array[Idx] << endl;
    }
}
```

The use of “`const`” preceding a pointer parameter specifies that the value of the target of the pointer cannot be modified by the called function. So, in the code above, `Print()` is forbidden to modify the value of the target of the pointer `Array`.

`Print()` also cannot modify the value of the actual pointer parameter since that parameter is passed by value.
3. Pointers

If “const int* iPtr” means that the TARGET of iPtr is to be treated as a const object, how would we specify that a pointer is itself to be a const?

```cpp
const int* const cPtr = new int(42);
```

Here, the value stored in the target of iPtr can be changed, but the address stored in iPtr cannot be changed. So, iPtr will always point to the same location in memory, but the contents of that location may change. (Array variables are essentially const pointers.)

Given the declaration of iPtr above:

```cpp
*iPtr = 17;  // legal
int anInt = 55;
iPtr = &anInt;  // illegal
```

Finally we can have a constant pointer to a constant target:

```cpp
const int* const cPtr = new int(42);
```

**Summary**

Courtesy of Bjarne Stroustrup, “The C++ Programming Language”

```cpp
void f1(char* p) {
    char s[] = "Gorm";  // pointer to char
    const char* pc = s;  // pointer to constant char
    pc[3] = 'g';          // error: target is constant
    pc = p;               // legal: pointer is malleable

    char* const cp = s;   // constant pointer
    cp[3] = 'g';          // legal: target is malleable
    cp = p;               // error: pointer is constant

    const char* const cpc = s; // constant pointer to constant target
    cpc[3] = 'g';          // error: target is constant
    cpc = p;               // error: pointer is constant
}
```

How to keep it straight? Stroustrup suggests reading the declarations backwards (right to left):

```cpp
char* const cp = s;
```

**cp is a constant pointer to a char**
3. Pointers

Pointer Array Arithmetic

If a pointer targets an array, it is possible to navigate the array by performing arithmetic operations on the pointer:

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

void main() {
    char s[] = "Gorm";
    char* p = s;
    for (int Idx = 0; Idx < strlen(s); Idx++, p++) {
        cout << setw(3) << Idx << "    " << *p << endl;
    }
}
```

produces the output:

```
0    G
1    o
2    r
3    m
```

Consider the update section of the for loop. At the end of each pass through the loop, we increment the value of the pointer `p`:

- `p++;` // increments the value of `p`
- `(*p)++` // increments the value of the target of `p`

The mystery here is: why does incrementing the value of `p` cause `p` to step through the array of characters, one-by-one?
### Array of Structs Pointer

```c++
#include <iostream>
#include <iomanip>
using namespace std;

struct Complex {
    double Real;
    double Imaginary;
};

void main() {
    const int SIZE = 5;
    Complex cArray[SIZE];
    Complex* cPtr = cArray;
    cout << "cPtr: " << cPtr << endl;
    cPtr++;
    cout << "cPtr: " << cPtr << endl;
}
```

**produces:**
```
cPtr: 006AFD78
```
```
cPtr: 006AFD88
```

Be very careful with code such as this….

…. the logic makes sense only if the target of the pointer is an array….

…. but, the syntax is legal no matter what the target of the pointer happens to be….

### Pointer Incrementing Abuse

```c++
#include <iostream>
#include <iomanip>
using namespace std;

void main() {
    double x = 3.14159;
    double* dPtr = &x;
    cout << " dPtr: " << dPtr << endl
        << " *dPtr: " << *dPtr << endl;
    dPtr++;
    cout << " dPtr: " << dPtr << endl
        << " *dPtr: " << *dPtr << endl;
}
```

**produces:**
```
dPtr: 006AFDC0
```
```
*dPtr: 3.14159
```
```
dPtr: 006AFDC8
```
```
*dPtr: 1.20117e-306
```

Incrementing `dPtr` makes no sense (logically) since that will simply make the target of `dPtr` the 8 bytes of memory that follow `x`. 
3. Pointers

Arrays of Pointers

Declarations:

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

Member Access

Field Access Examples:

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

Arrow Operator

Short-hand notation:

```c
cout << rayPtrs[size-1]->field1
    << rayPtrs[size-1]->field2
    << rayPtrs[size-1]->field3;
```

Using the same sorting algorithm, why is sorting an array of pointers to records faster than sorting an array of records?

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:

```
before
  *iptr1
during
  iptr1 = NULL;
after
  *iptr1
```

Aliases

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

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

Dangling Pointers

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

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

Memory leaks
3. Pointers

Reference Variables

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

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

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:

```c
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.

Reference Returns

Return by Value
Normally most function returns are by value:

```c
int f(int a) {
  int b = a;
  // ... return( b );
} //f
```

The function does not actually return b, it returns a copy of b.

Return by Reference
Functions can return references:

```c
int& f(int& a) {
  int b = a;
  // ... return( b );
} //f *** bad ***
```

Good compilers will issue a warning for returning a reference to a local variable.

```c
int& f(int& a) {
  int b = a;
  // ... return( a );
} //f *** alias ***
```

Do NOT return references to private data members of a class. This violates the encapsulation of the class.