Slides

Dynamic Variables

Static Variables

- Size is fixed throughout execution
- Size is known at compile time
- Space/memory is allocated at execution

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)

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

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,
      fptr2;         <- not a pointer
```

common declaration

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

(iptr2: not a pointer)

- 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)**

Address Operator: & (ampersand)

- 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: * (asterisk)

- 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;
```

- (Possible) results:

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

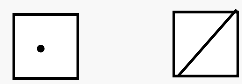
NULL Pointer

- Pointer constant, address 0

- Named constant in the `<cstddef>` include header (`<stddef.h>` old style header).

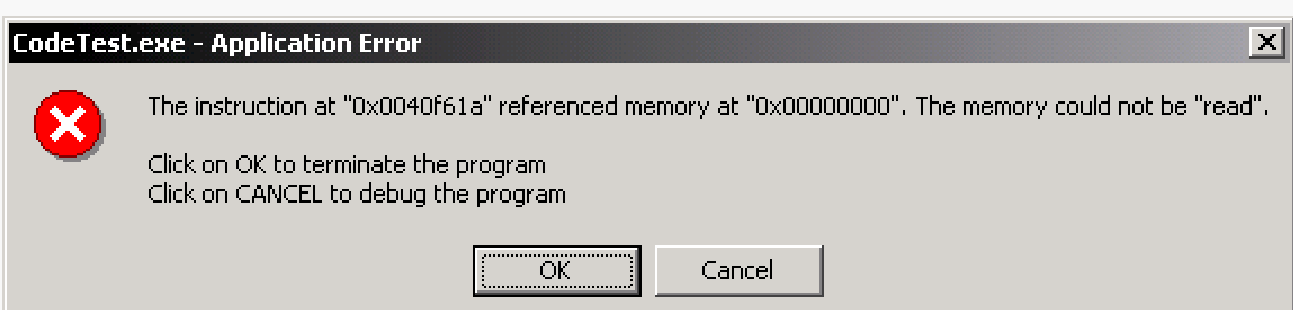
- Represents the empty pointer
  - † points nowhere , unique pointer/address value

- Symbolic/graphic representations:



- Illegal: NEVER dereference a pointer that equals NULL

`*NULL`

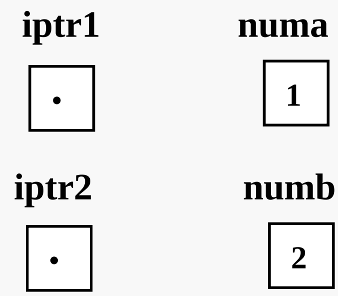


Pointer Diagrams

– Given (text/code representation) representation

```
#include <cstddef>
void main() {

   int* iptr1 = NULL;
   int* iptr2 = NULL;
   int  numa, numb;

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

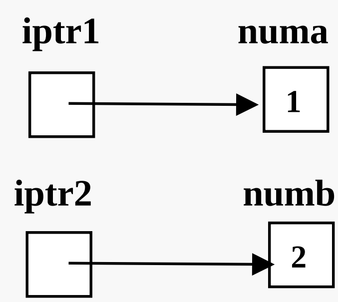
Graphic



Pointer Assignments

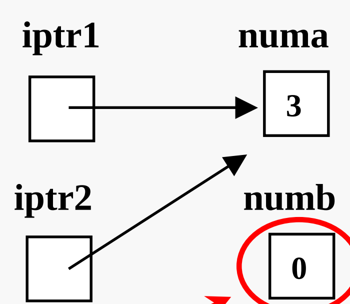
– #1

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



– #2

```
*iptr2 = *iptr1 - 1;
iptr2  = iptr1;
*iptr2 = 3 ;
```



No pointer access to numb remains.

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.

```
. . .
  int* iptr1 = NULL;
  int* iptr2 = NULL;
. . .
```

- 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:

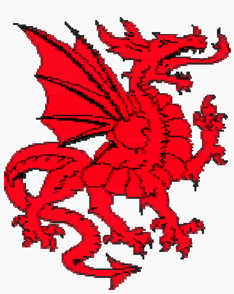
```
iptr2  = *iptr1; // error: assign int to int*
```

or vice versa:

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

Typecasts and pointers:

- the assignments above would be legal if an explicit typecast were used:



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

```
typedef int* iPtr;
iptr2  = iPtr(*iptr1);    // legal
```

```
*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.

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
0xF4242	???
• • •	• • •

Record Pointers

Pointers to structures:

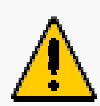
– Given:

```
const int f3size = 20;

struct rectype {
  int   field1;
  float field2;
  char  field3[f3size];
};

typedef rectype *recPtr;

rectype rec1 = {1, 3.1415f, "pi"};
recPtr  r1ptr;

r1ptr = &rec1;
```

Member Access

– Field Access Examples:

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



Note: parentheses are 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

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**

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

Allocation

```
char* name;
int*  iptr;
//C++
name = new(nothrow)char;


iptr =
   new(nothrow) int [20];

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

pointer typecasts required

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

iptr = (int *)
malloc(20*sizeof(int));
```

dynamic array allocation

Deallocation

```
//C++
delete   name;
name = NULL;
delete [] iptr;
//delete [20] iptr;
iptr = NULL;
```

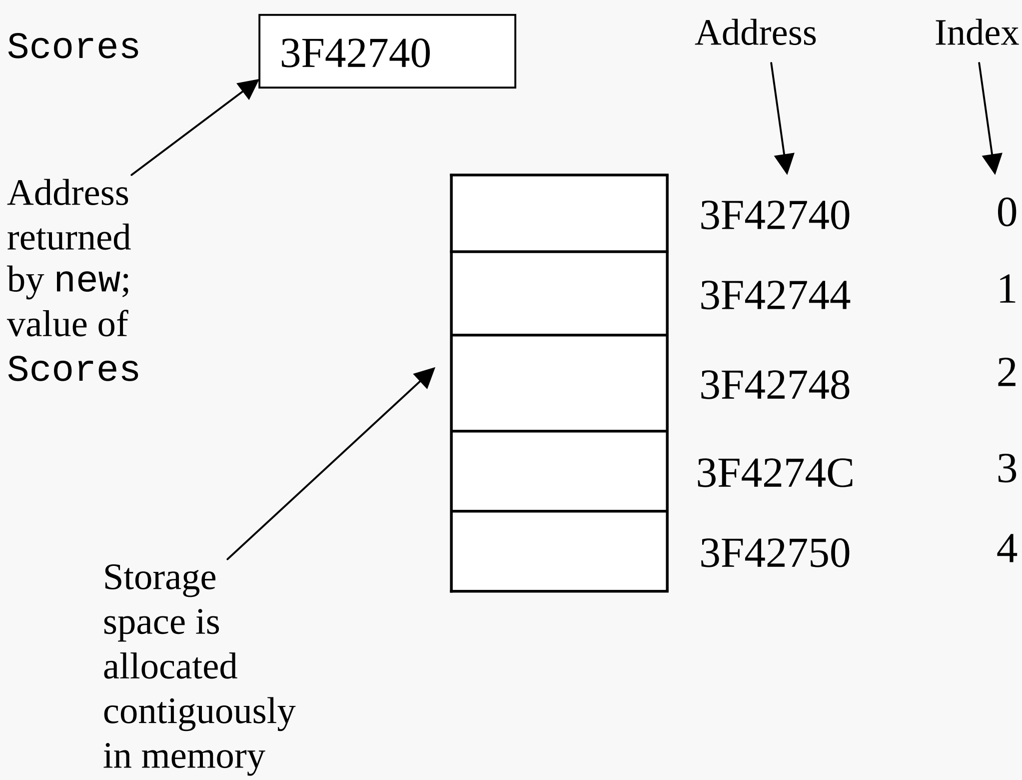
```
// C
free(name);

free(iptr);
```

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

Declaration Syntax

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

Effect of array allocation via new

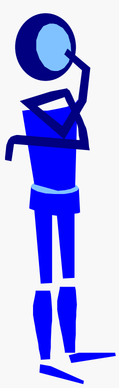


Use like any statically-allocated array

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

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

Deallocation

```
delete [] Name;
delete [] Scores;
delete [20] Scores;// including dim is optional
                           // and has no effect
```



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.

```
// WARNING
delete Name;

//May not release array memory, undefined results
```

Resizing a dynamically-allocated array

```
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:

```
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.

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

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

void main() {

   int  Count;
   int* t;
   const int Size = 900000000;
   int* myList = new(nothrow) int[Size];

   if (myList == NULL) {
      cout << "Allocation failed!!" << endl;
      return;
   }

  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;
  }

}
```

What if `t` was replaced with `myList`?

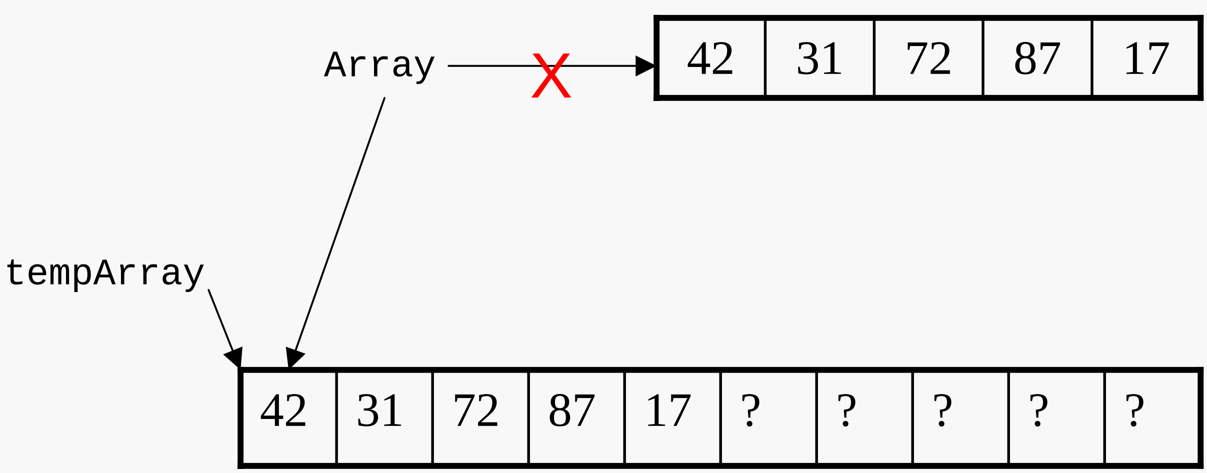
Reference Pointer Parameters

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:

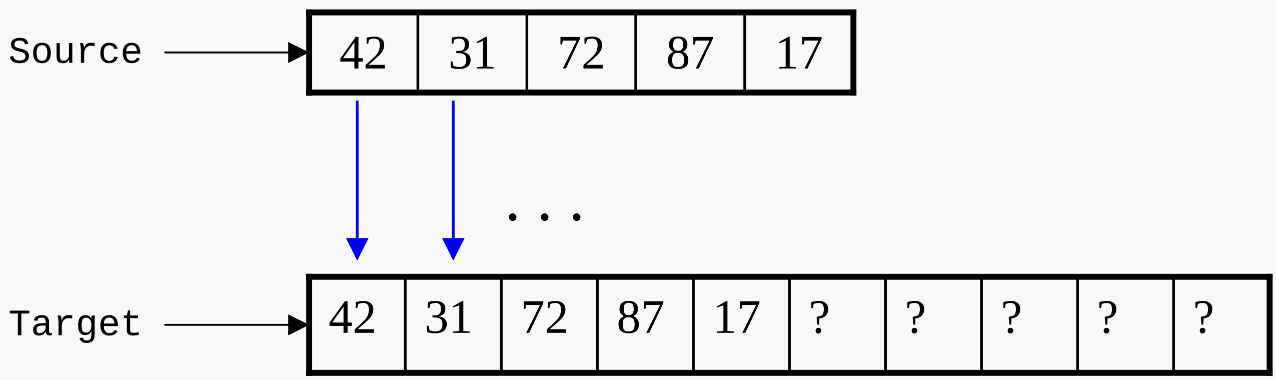
This pointer is being passed by reference.

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



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

```
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).

```
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?

```
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.

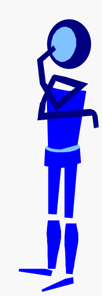
const Pointers

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`?

```
// constant pointer to int
int* const iPtr = 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.

Given the declaration of `iPtr` above:



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

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

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

const Summary

Courtesy of Bjarne Stroustrup, "The C++ Programming Language"

```
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):

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

cp is a constant pointer to a `char`

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?

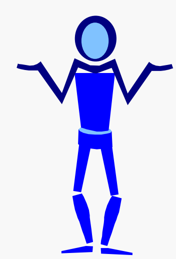
Incrementing Pointers

From B. Stroustrup, "The C++ Programming Language":

The result of applying the arithmetic operators `+`, `-`, `++`, or `--` to pointers depends on the type of the object pointed to. When an arithmetic operator is applied to a pointer `p` of type `T*`, `p` is assumed to point to an element of an array of objects of type `T`; `p+1` points to the next element of that array, and `p-1` points to the previous element. This implies that the integer value of `p+1` will be `sizeof(T)` larger than the integer value of `p`.

In other words, the result of incrementing a pointer depends on the type of thing to which it points.

```
const int SIZE = 5;
int iArray[SIZE] = {32, 17, 89, 43, 91};
int* iPtr = iArray;

for (int k = 0; k < SIZE; k++, iPtr++)
   cout << setw( 3) << k
        << setw(10) << iPtr
        << setw(10) << *iPtr
```



produces:

```
  0   006AFDD0        32
  1   006AFDD4        17
  2   006AFDD8        89
  3   006AFDDC        43
  4   006AFDE0        91
```

Why does this output make sense?

Array of Structs Pointer

```
#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

```
#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`.

Arrays of Pointers

Declarations:

– Given:

```
const int size = 20;

struct  rectype {
  int   field1;
  float field2;
  char  field3[size];
};
typedef  rectype  *recPtr;

rectype  rec1 = {1, 3.1415f, "pi"};
recPtr   rayPtrs[size];

rayPtrs[size-1] = &rec1;
```

Member Access

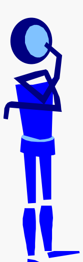
– Field Access Examples:

```
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;
```



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

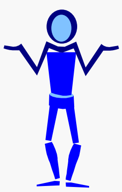
Dynamic Memory Problems

Given:

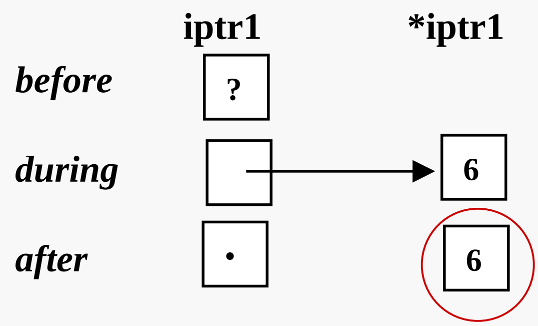
```
typedef int *intPtr;
intPtr  iptr1, iptr2;
```

Garbage

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



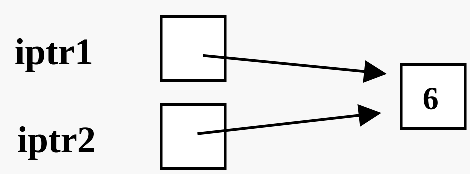
```
iptr1 = new int (6);
iptr1 = NULL;
```



Aliases

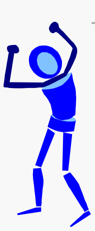
- Two or more pointers referencing the same memory location.
- Example:

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

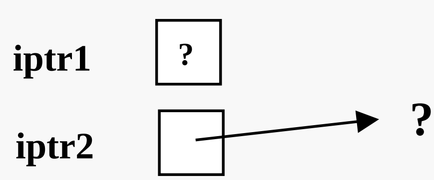


Dangling Pointers

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



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



memory leaks

Reference Variable Declarations

- The ampersand ‘&’ 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.

Return by Value

Normally most function returns are by value:

```
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:

```
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.

The code above contains a subtle trap. The function returns a reference to a variable b which will no longer exist when the function exits and goes out of scope. Returning a reference to an already referenced variable is acceptable, (although most likely unnecessary and confusing).

```
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.