Constructors and Const
- Remember some methods are declared constant?
  ```cpp
template<
  void getObjectId() const;
  void CreateAndDestroy::getObjectID() const
  {
    return objectID;
  }
```
- The "constness" of an object begins as soon as the constructor finishes its job and ends when the destructor is called.
- How do we initialize constant member data?

Example
```cpp
class MyClass {
public:
  MyClass(int initial_x);
private:
  const int x; //how do we initialize this?
}
```

Const Problem
- How do you initialize const data when const data has to be initialized when it is declared?
- Through the use of a member initializer list
  ```cpp```
  MyClass::MyClass(int initial_x) : x(initial_x)
  {
    //whatever else the constructor needs to do
    //we can’t do an ‘x=initial_x’ because x is const!
  }
```

Const Problem Solved
- Can use this to initialize all member data
- *Must* use this to initialize const member data

Reference Variable
- The ampersand ‘&’ character is used for reference variable declarations:
  ```cpp
  int &iptr;
  float &fptr1, &fptr2;
  ```
- Reference variables do NOT use the address and dereference operators ‘& ‘.
- Compiler dereferences reference variables transparently.
Reference Variable

- Reference variables are constant addresses, assignment can only occur as initialization or as parameter passing, reassignment is NOT allowed.
- 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

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

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

Functions can return references:

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

– 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).

Watch Out

- Returning a reference (like a pointer) to a private member
  – Will not work! Why?
- This would break the information hiding, so the compiler will not allow it!
- E.g.:
  ```c
  class MC {
  public:
    int& getX() {return &x;}
  private:
    int x;
  }
  ```

Const Pointers as Function Parameters

- Four ways to use const with pointers and functions
  – Non-constant pointer to non-constant data
    • Data can be manipulated and pointer can change
  – Non-constant pointer to constant data
    • Pointer can be made to point to something else, data cannot change
  – Constant pointer to non-constant data
    • Data can be changed, pointer cannot
  – Constant pointer to constant data
    • Nothing can change
  – ☺ Simple, Right?

Examples

//pointer to a const char
void printCharacters ( const char *sPtr )
{
    for ( ; *sPtr != '0'; sPtr++ )
        cout << *sPtr;
} //The pointer is changing; data is not
More Examples

```c
int main()
{
    //Can you see the error?
    int x, y;
    int * const ptr = x; //const pointer to an int
    *ptr = 7;
    ptr = &y;
    return 0;
}
```

One more example

```c
int main()
{
    //Are there errors?
    int x=5, y;
    const int * const ptr = &x; //const pointer to a const int
    *ptr = 7;
    ptr = &y;
    return 0;
}
```

Pointer Expressions and Pointer Arithmetic

- You can:
  - Increment
  - Decrement
  - Add
  - Subtract
  - Compare
- +1 adds the size of the type
- E.g. if an int was 4 bites, and an int ptr pointed to 0, ++ptr would point to bite number 4

void *

- void * is a pointer to any type of data
- It should be avoided unless necessary
- A pointer of any type can be cast to a void *
- You cannot dereference a void *
- You must first cast the void * to the type of pointer it is, then dereference

Array Pointer

- Assume we have int b[6] and int * bPtr
- We can do this:
  - bPtr = b; //an array is a pointer!
  - bPtr = &b[ 0 ]; //array points to first element
- Also, for example b[3] is:
  - *( bPtr + 3 )
  - *( b + 3 )
- What does “cout << *b;” print?

Arrays of Pointers

- Consider the following declaration:
    “Clubs”, “Spades”
  };
- How does this compare with a two dimensional array that would normally have to hold these strings?
  - What about in memory?
Dynamic Data ☺

- The programmer can if they wish create an object dynamically.
- Meaning rather than using memory given to the program when it begins and resides in its memory space.
- You can get it from the system heap.
- The new keyword!

Syntax

- You use the command new
- Time *timePtr;
- timePtr = new Time; //Which constructor?
- timePtr2 = new Time(1,2,1980);
- New returns a pointer to the memory allocated for the newly created object of type Time.
- #include <new> to use the new standard

More examples

- You can do this for any built-in or user-defined type.
- int *xPtr = new int;
- You can also create an array this way
- int xArrayPtr = new int [10];
- This creates an array of size ten and you access it through xArrayPtr
- cout << xArrayPtr[1];

Freeing memory

- To release the memory pointed to by your pointer you use the command delete
- delete xPtr;
- delete [] xPtr;
- Forgetting the [] on an array only releases the memory for the first location in the array!!!

Pointers to structures:

```cpp
const int f3size = 20;

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

typedef rectype *recPtr;
recPtr rec1 = {1, 3.1415f, "pi"};
recPtr rlptr = &rec1;
```

Logical Expressions for Pointers

- NULL tests:
  ```cpp
  if (!person) //true if (person == NULL)
  ```

- Equivalence Tests:
  ```cpp
  if (person == name) //true if pointers reference the same memory address
  //person and name are pointers
  ```
Deallocation

- 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.

“Growing” an 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;
```

nothrow

- 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.

More nothrow

- the C++ 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];
```

- What should you always do after declaring a new object?
- CHECK TO SEE IF IT WAS NULL!

Pointers Passed

- Passed by value
  - When they do not need to change the pointer value itself
- Passed by reference
  - When they change what the pointer is pointing to
- void add(node *&list, int val) {
  //add is a pointer here
}

Dynamic Memory Problems

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

```
before
iptrl  *iptrl

during
iptrl = new int (6);

after
iptrl = NULL
```
Dynamic Memory Problems

- **Aliases**
  - Two or more pointers referencing the same memory location.
  - Example:
    
    ```cpp
    iptr1 = new int (6);
    iptr2 = iptr1;
    ```

- **Dangling Pointers**
  - Pointers that reference memory locations previously deallocated.
  - Example:
    
    ```
    iptr1 = new int (6);
    iptr2 = iptr1;
    delete iptr1;
    ```

Mechanism for deep copy

- Solution is to provide what is know as a mechanism for a deep copy
- When you have dynamic data inside of a class, you should always supply three methods
  1. Copy Constructor
  2. Assignment Operator (overloaded =)
  3. Destructor

Copy Constructor

- A copy constructor allows you to successfully create an object that is a copy of another
- e.g. Student NewStudent = OldStudent;
- This would invoke the copy constructor.
- The copy constructor would take care of creating and copying the course information
- Note that NewStudent didn’t exist before the statement above

```cpp
Student::Student(const Student& RHS)
{
    //you perform a memberwise copy
    CoursePtr = new Course[size];
    for ( int i=0; i<Used; i++ )
    {
        CoursePtr[i] = RHS.CoursePtr[i];
    }
} //student has an array of courses
```
Assignment Operator

- An assignment operator allows you to transfer a copy of an already existing object into an already existing object.
- e.g. StudentA = StudentB;
- This is a simple assignment statement.
- The difference between this and a copy constructor is the missing StudentA already exists.

const Student& Student::operator=(const Student& RHS )
{
    if ( this != &RHS )
    {
        delete [] this.CoursePtr;
        //perform memberwise copy
        CoursePtr = new Course [size];
        for ( int i=0; i<Used; i++ )
        {
            CoursePtr[i] = RHS.CoursePtr[i];
        }
        return *this;
    }  //Why is a "const Student&" returned?
    x=y=z;//y=z must return a Student& to be assigned to x
}

Destructor

Student::~Student()
{
    delete [] CoursePtr;
}