

USING POINTERS

Today

- Today we will look at topics relating to the use of pointers
 - Call by value
 - Call by reference
 - Copy constructors
 - Dynamic memory allocation
- The last of these isn't directly to do with pointers, but we need to cover it to understand some of the stuff that is to do with pointers.
- This material is kind of covered in Chapter 3 by Pohl.
- All the examples in these notes are on the class website.
- We will start by recalling the use of functions.

Functions: parameters and arguments

- Function header declaration:

```
type name ( parameters );
```

- Function definition:

```
type name ( parameters ) {  
    statements  
}
```

- Function invocation:

```
name ( arguments );
```

or

```
variable_of_type = name ( arguments );
```

- Functions have to be declared before they can be called

- The book uses the word “parameters” when a function is declared and “arguments” when a function is invoked (or “called”)
- When a function is called, the program control shifts from wherever the function call originates to the body of the function
- The function arguments get initialized as local variables within the function.
- Now, parameters can be either:
 - *call by value* or
 - *call by reference*

Call by value

- With *call by value*, the *value* of each argument is copied to a local variable within the function
- When the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- Even if the values of the local arguments within the function changed during the execution of the function, the values that were used to invoke the function do not change

- Example:

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

void myfun( int a ) {
    a++;
    cout << "inside myfun, a=" << a << endl;
} // end of myfun()

int main() {
    int a = 7;
    cout << "before calling myfun, a=" << a << endl;
    myfun( a );
    cout << "after calling myfun, a=" << a << endl;
} // end of main()
```

- The output is:

```
before calling myfun, a=7
inside myfun, a=8
after calling myfun, a=7
```

Call by reference

- With *call by reference*, the *address* of each argument is copied to a local variable within the function
- When the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- Because the local arguments are addresses, any changes that were made to the values stored at these address locations during the execution of the function *are retained* when the function ends
- in C++, there are two ways to implement call by reference:
 - using pointers; and
 - using references.

- Example of call by reference using pointers:

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

void myfun( int *a ) {
    (*a)++;
    cout << "inside myfun, *a=" << *a << endl;
} // end of myfun()

int main() {
    int a = 7;
    cout << "before calling myfun, a=" << a << endl;
    myfun( &a );
    cout << "after calling myfun, a=" << a << endl;
}
```

- And the output is:

```
before calling myfun, a=7
inside myfun, *a=8
after calling myfun, a=8
```

- Thus pointers give us one way of “reaching” things outside functions.

- Example of call by reference using references:

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

void myfun( int &a ) {
    a++;
    cout << "inside myfun, a=" << a << endl;
} // end of myfun()

int main() {
    int a = 7;
    cout << "before calling myfun, a=" << a << endl;
    myfun( a );
    cout << "after calling myfun, a=" << a << endl;
}
```

Why use call-by-reference?

- We use call-by-reference for *efficiency*.
- Call-by-value requires the computer to copy the parameters before passing them to the function.
- This is fine if the parameters are a few chars or doubles.
- But in C++ we might call a function on a complex object that holds many many bytes of data.
- It is far more efficient, in both memory and time, to pass a pointer or a reference to such an object than to copy it.
- However, you have to be very careful when you do this otherwise you may get odd things happening to your program.

Copy constructors

- If you do decide to pass a complex object by call-by-value, you need to define a *copy constructor* for it.
- The problem is that C++ on its doesn't know how to copy complex objects.
- So you have to describe exactly how to make a copy.
- Here's a copy constructor for the `point` object:

```
point::point(const point& p) {  
    x = p.x;  
    y = p.y;  
}
```

- (`point` is not complex enough to require a copy constructor, but it makes a good example since we know it so well by now).

- C++ knows this is a copy constructor by the signature.
- There is no return type (just like a constructor).
- The only argument is a reference to an object of the same class as the constructor is defined for.
- The `p` that is the argument of the copy constructor is the object being copied.
- What the copy constructor has to do is to say how to set the value of every attribute of the object.
- In the example from `point`, we are saying that to make a copy of `p` copy the attribute `p.x` into the attribute `x` of the copy, and similarly for `y`.

- Using a copy constructor we get a *deep copy* of the original object.
- This is in contrast to the *shallow copy* that we get if we don't define a copy constructor.
- Roughly speaking, if an object includes a pointer, we need to make a deep copy of the object.
- For a more complex example of a copy constructor, see the example program `dynamic-stack.cpp`.

Passing arrays to functions

- Given the following example:

```
int sum( int A[], int n )  
{  
    int s=0;  
  
    for ( int i=0; i<n; i++ )  
        s += A[i];  
    return( s );  
} // end of sum()
```

- When the array `A` is passed to the function `sum()`, it is passed using call-by-value on its base address (i.e., the address of `A[0]`)
- However, passing an address call-by-value is the same as passing the thing that is addressed call-by-reference.

- Thus within the context of a function header definition, the following two statements are equivalent:

```
int sum( int A[], ... ) { ... }
```

and

```
int sum( int *A, ... ) { ... }
```

but not in other contexts!

- This explains the function headers you see in some of the C++ libraries.

- And the output is:

```
before calling myfun, a=7
inside myfun, a=8
after calling myfun, a=8
```

Generic pointers

- Last class, we talked about pointers to specific data types, e.g.,:

```
int *pToInt, *pToInt2;
char *pToChar;
```

- You can also have a pointer to a void:

```
void *pToVoid;
```

- Clearly this is not a pointer *to* anything (what is a void?).
- A “pointer to a void” is a *generic* pointer.
- You can use it to point to different kinds of object.
- When you *dereference* the pointer, it is like converting it to that data type

- Below are all legal statements, given the definitions above:

```
pToVoid = pToInt;
pToInt2 = reinterpret_cast<int*>(pToVoid);
pToChar = &C;
pToVoid = pToChar;
pToVoid = &A;
```

But you can't do this:

```
pToInt2 = pToVoid;
```

- See `generic-pointer.cpp`
- You can use a generic pointer, for example, as an argument to a function to which you might need to pass different kinds of object.

Dynamic memory allocation

- In C++, there are two functions that handle dynamic memory allocation: `new` and `delete`
- The syntax for `new` is:
`new type-name`
`new type-name initializer`
`new type-name[expression]`
- For example:

```
int *p, *q, *r;  
p = new int(5); // allocation and initialization  
q = new int[10]; // allocation, but uninitialized  
r = new int; // allocation, but uninitialized
```
- Some compilers initialize values to 0 by default, but not all—that is not part of the language specification, so don't rely on it!

- The syntax for `delete` is:
`delete expression`
`delete[] expression`

- The first form is for non-arrays; the second form is for arrays
- The point of dynamic memory allocation is to allow your program to decide, while running, how much data it needs to store.
- You can, therefore, tailor the size of an array to the problem you are trying to solve.
- The next slide gives an example (from book, p137).
- Other examples can be found in the two stack handling programs `stack-with-ctors.cpp` (from Unit II) and `dynamic-stack.cpp`.

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int *data;  
    int size;  
  
    cout << "\nenter array size: ";  
    cin >> size;  
    assert( size > 0 );  
  
    data = new int[size]; // allocate array of ints  
    assert( data != 0 );  
  
    for ( int j=0; j<size; j++ ) {  
        cout << (data[j]=j) << '\t';  
    }  
    cout << "\n\n";  
    delete[] data; // deallocate space for array  
  
} // end of main()
```

- In general, pointers go well with dynamic memory allocation.
- If you don't know how often you will call `new`, then you can't specify the size of an array, and you can't give every new piece of allocated memory a name.
- But you can have a pointer that knows its location in memory.

```
int *pToInt;  
  
pToInt = new int;
```

- To keep track of lots of dynamically allocated memory, we often created linked datastructures, like that in `dynamic-stack.cpp`.
- Other structures like this are covered in CIS 22 Datastructures.

Namespaces

- You have already been using namespaces as in:

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

- The `std` namespace is the standard C/C++ namespace that comes with the language
- A namespace is a way of grouping classes to avoid name conflict
- That is, you could have two things with the same name, but in different name spaces, and then there would be no conflict

- Declaration of classes within a namespace looks like this:

```
namespace myspace {

    class myclass1 { ... };

    class myclass2 { ... };

} \\ end of namespace
```

- Note that when you define a namespace in a header file, you do not need to use the `.h` in the include statement:

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

versus

```
#include <time.h>
```

- The first include statement is part of a namespace; the second is not

Summary

- This lecture has looked at uses of pointers:
 - We mainly discussed the use of pointers (and references) for call-by-reference.
 - We also described copy constructors;
 - Generic pointers; and
 - Dynamic memory allocation.
- Finally, we briefly discussed namespaces.