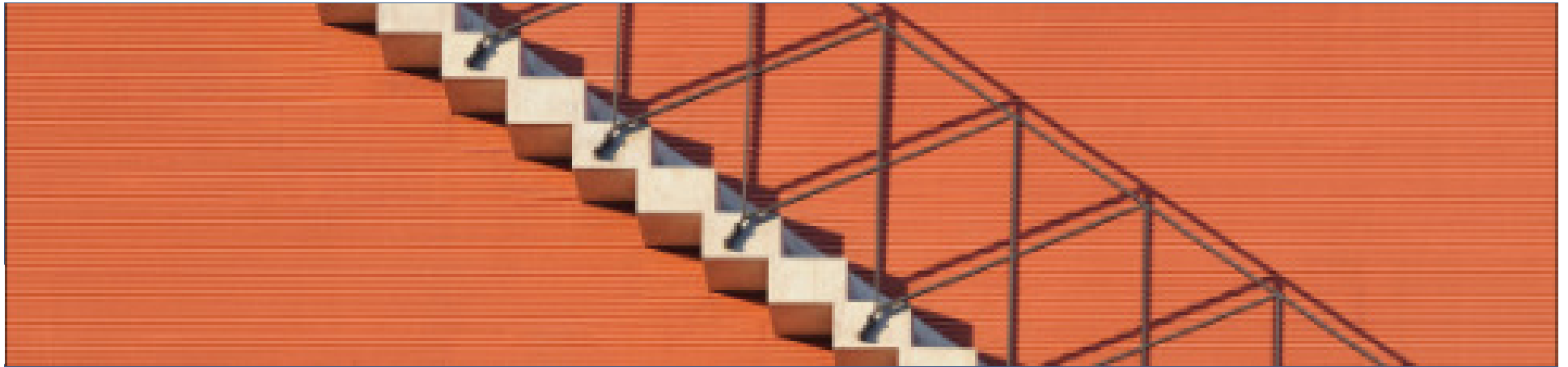




Module 6

- Pointers
- void*

Objectives

- In this chapter, you will:
 - Use the pointer data type and pointer variables
 - Declare and manipulate pointer variables
 - Learn about the address of operator and the dereferencing operator
 - Discover dynamic variables
 - Use the **new** and **delete** operators to manipulate dynamic variables
 - Learn about pointer arithmetic
 - Discover dynamic arrays

Declaring Pointer Variables

- Syntax:

```
dataType *identifier;
```

- Examples:

```
int *p;  
char *ch;
```

- These statements are equivalent:

```
int *p;  
int* p;  
int * p;
```

Declaring Pointer Variables (cont'd.)

- In the statement:

```
int* p, q;
```

- Only `p` is a pointer variable
- `q` is an `int` variable

- To avoid confusion, attach the character `*` to the variable name:

```
int *p, q;
```

```
int *p, *q;
```

Address of Operator (&)

- Address of operator (&):
 - A unary operator that returns the address of its operand

- Example:

```
int x;  
int *p;  
p = &x;
```

- Assigns the address of `x` to `p`

Dereferencing Operator (*)

- Dereferencing operator (or indirection operator):
 - When used as a unary operator, * refers to object to which its operand points

- Example:

```
cout << *p << endl;
```

- Prints the value stored in the memory location pointed to by p

structs and Pointer Variables

- You can declare pointers to other data types:

```
struct studentType
{
    char name[26];
    double gpa;
    int sID;
    char grade;
};
```

```
studentType student;
studentType *studentPtr;
```

- student is an object of type studentType
- studentPtr is a pointer variable of type studentType

structs and Pointer Variables (cont'd.)

- To store address of `student` in `studentPtr`:
`studentPtr = &student;`
- To store 3.9 in component `gpa` of `student`:
`(*studentPtr).gpa = 3.9;`
 - () used because dot operator has higher precedence than dereferencing operator
 - Alternative: use **member access operator arrow** (`->`)

structs and Pointer Variables (cont'd.)

- Syntax to access a `struct` member using the operator `->` :

```
pointerVariableName->classMemberName
```

- Thus,

```
(*studentPtr) . gpa = 3.9;
```

is equivalent to:

```
studentPtr->gpa = 3.9;
```

Initializing Pointer Variables

- C++ does not automatically initialize variables
- Pointer variables must be initialized if you do not want them to point to anything
 - Initialized using the **null pointer**: the constant value 0
 - Or, use the `NULL` named constant
 - The number 0 is the only number that can be directly assigned to a pointer variable

Dynamic Variables

- Dynamic variables: created during execution
- C++ creates dynamic variables using pointers
- `new` and `delete` operators: used to create and destroy dynamic variables
 - `new` and `delete` are reserved words in C++

Operator new

- new has two forms:

```
new dataType;           //to allocate a single variable  
new dataType[intExp];   //to allocate an array of variables
```

- intExp is any expression evaluating to a positive integer
- new allocates memory (a variable) of the designated type and returns a pointer to it
 - The allocated memory is uninitialized

Operator new (cont'd.)

- Example: `p = new int;`
 - Creates a variable during program execution somewhere in memory
 - Stores the address of the allocated memory in `p`
- To access allocated memory, use `*p`

Operator delete (cont'd.)

- **Memory leak:** previously allocated memory that cannot be reallocated
 - To avoid a memory leak, when a dynamic variable is no longer needed, destroy it to deallocate its memory
- `delete` operator: used to destroy dynamic variables
- Syntax:

```
delete pointerVariable;    //to deallocate a single
                           //dynamic variable
delete [] pointerVariable; //to deallocate a dynamically
                           //created array
```

Operator `delete` (cont'd.)

- `delete` only marks the memory space as deallocated
 - Pointer variable may still contain address of deallocated memory space
- If you try to access via the pointer, could result in corrupting data or termination
- Avoid this by setting pointers to `NULL` after delete operation

Operations on Pointer Variables

- Can Perform pointer arithmetic
- Pointer arithmetic can be very dangerous:
 - Program can accidentally access memory locations of other variables and change their content without warning
 - Some systems might terminate the program with an appropriate error message
- Always exercise extra care when doing pointer arithmetic

Dynamic Arrays

- Dynamic array: array created during program execution

- Example:

```
int *p;
```

```
p = new int[10];
```

```
*p = 25;
```

```
p++; //to point to next array component
```

```
*p = 35;
```

← stores 25 into the first memory location

← stores 35 into the second memory location

Dynamic Arrays (cont'd.)

- Can use array notation to access these memory locations
- Example:

```
p[0] = 25;  
p[1] = 35;
```

 - Stores 25 and 35 into the first and second array components, respectively
- *An array name is a constant pointer*

Functions and Pointers

- Pointer variable can be passed as a parameter either by value or by reference
- As a reference parameter in a function heading, use &:

```
void pointerParameters(int* &p, double *q)
{
    . . .
}
```

Pointers and Function Return Values

- A function can return a value of type pointer:

```
int* testExp(...)  
{  
    . . .  
}
```

You will find very quickly that these * floating around get very confusing. It's better to create a **pointer type** alias.

For example,

```
typedef DataType* DataPtr;
```

and then use the type to create pointer variables

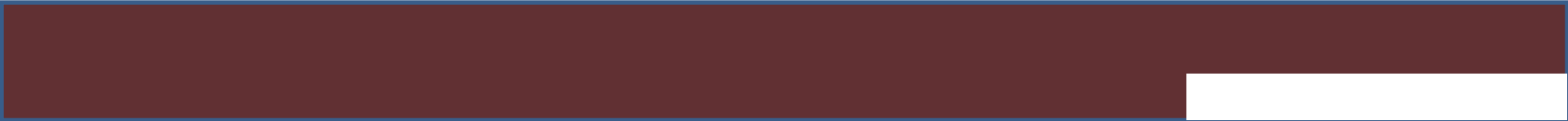
```
DataPtr Index[10];
```



This is useful for passing pointers to functions, especially when passing by reference.

Passing by reference is just the placing of the address of the argument on the stack.

That is, the formal argument of a function is an alias to the actual argument.



Consider the swap function for exchanging two integer values. In C++ this is:

```
void Swap(int& X1, int& X2)
{
    int temp;

    temp = X1;
    X1 = X2;
    X2 = temp;
}
```

The addresses of the actual arguments A and B are put on the stack and used as the addresses of X1 and X2 in the function.

and would be called, for example, as

```
Swap (A, B) ;
```

In C, the addresses must be explicitly put on the stack

```
Swap (&A, &B) ;
```

and the function must reference these addresses as pointers and dereference them to change the data pointed at.

```
void Swap(int *X1, int *X2)
{
    int temp;

    temp = *X1;
    *X1 = *X2;
    *X2 = temp;
}
```

We are not changing the pointers, just what they point at.

Let's talk about passing array arguments. We all know that arrays are passed by reference

- the address of the **first** element is put on the stack.

Consider

```
int SumArray(int arr[], int n)
{
    int i, sum=0;

    for (i=0;i<n;i++)
        sum += arr[i];
    return sum;
}
```



For the array

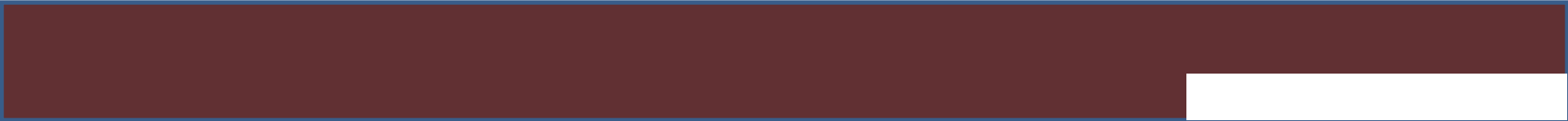
```
int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

we can sum the entire array as

```
SumArray(A, 10)
```

or

```
SumArray(&A[0], 10)
```

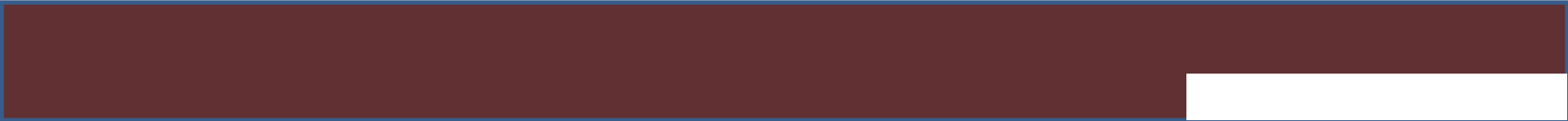




But the same function can also sum the last nine elements using

```
SumArray(&A[1], 9)
```

That is, an array name and an address seem to be equivalent.





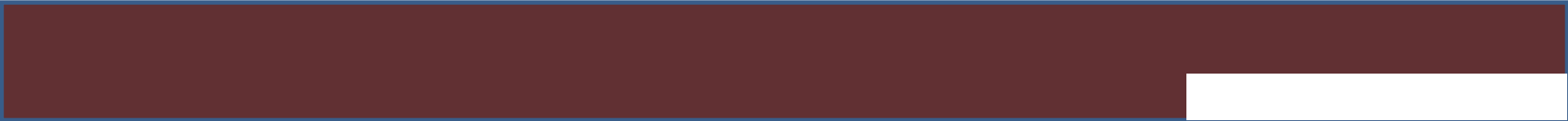
In fact, a pointer type can also be referenced like an array.

So

```
int A[10];  
int* B = A;
```

means the pointer `B` gets the address of the array `A` (its name).

Now `B` can be used just like `A` could.



When a C++ program references array elements, the compiler has to do some **pointer arithmetic**.

For example, $A[1]$ refers to the memory location one after the address A .

In pointer arithmetic this is $\ast(A + \mathbf{1})$.

One what? **One** memory location.

What's that? Depends on type of A .

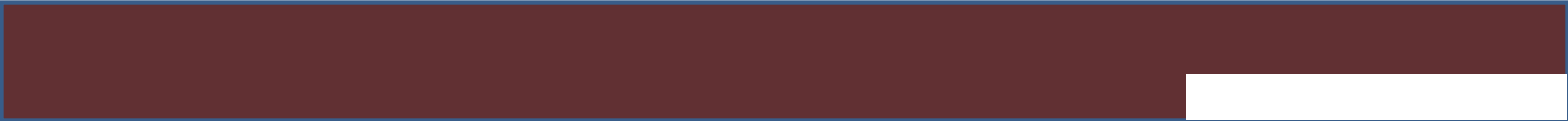


Although C++ will use the right size to perform the pointer arithmetic, sometimes we like to know how much space types use.

C++ provides an operator called `sizeof` to give the programmer this information.

The operator usually appears looking like a function as in

```
sizeof(int)
```



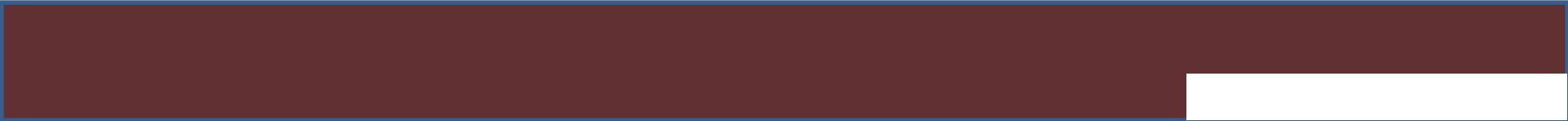


It will even tell us the size of an array.

```
const char message[] = "How long is this?";
```

would produce 18 for

```
sizeof(message)
```





Consider

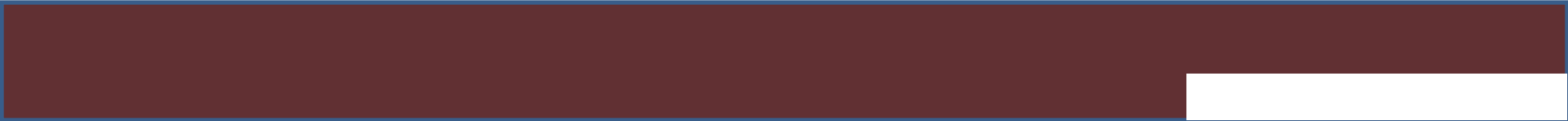
```
void SumArray(int arr[], int n)
```

Why can't we just leave out the second argument and use

```
n = sizeof(arr);
```

within the function?

Because all we know about `arr` is its address – stored in an `int*` – size 4 bytes.





There are very few operators for pointers:

++

--

+ an integer

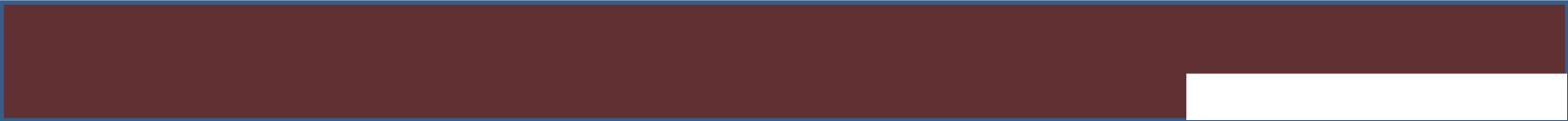
+= an integer

- an integer

-= an integer

- another pointer

-= another pointer



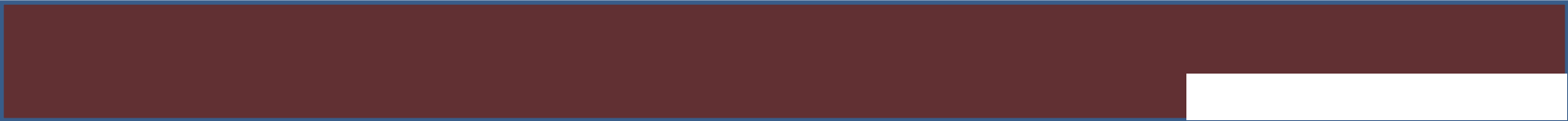


A pointer requires a value which is the address of a data value.

This may be the address of a constant, a global variable, an automatic variable or a local variable.

It can be passed as an argument, both by reference and by value.

It can also be the return value of a function.





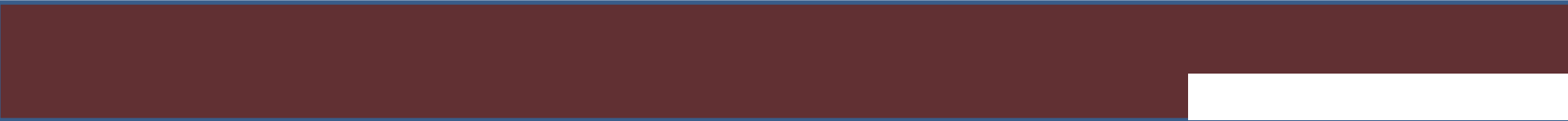
We have already seen functions which return pointer values although we probably haven't used them as such.

For example

```
char* strcpy (char*, const char*);
```

returns a pointer to the destination array.

But other functions in the string library require us to use the pointer value.



Get used to deleting dynamically allocated memory once their usage is complete.

Avoid memory leaks.

```
void func()  
{  
    int* ptr;
```

automatic variable

```
    ptr = new int[100];  
    . . .  
}
```

ptr is destroyed
100 ints leaked
unless we delete [] ptr

void Pointers (cont'd)

The C approach to building **generic data types** is to use a special type of pointer which can represent any pointer type.

➔ It is called the *void pointer*.

We declare a void pointer using

void* varname ;

which can hold an address of a variable of **any** data type.

void Pointers (cont'd)

- Because the compiler does not know the type of data a `void*` pointer is pointing at, the only thing we can do with such a pointer is transfer it – by assignment, copying and function arguments.
- What we **cannot do** is directly dereference it because it does not point at any specific type.
- We have to **type cast** it before dereferencing it.

void Pointers (cont'd)

The cast

`(type *) vptr` or

`static_cast <type *> (vptr)`

will convert the void pointer `vptr` to a type pointer.

Here's an example.

`int *ip;`

`void *vp;`

- pointers of two types.

void Pointers (cont'd)

```
int *ip, num=10;  
void *vp;  
ip = new int;  
*ip = num;  
vp = ip; // the cast to void* is automatic
```

```
cout << *vp << endl; // compile error  
cout << *(static_cast<int*>(vp)) << endl; // ok prints int
```

C++ doesn't know how to print out a void type !!!
But it can print an `int`.

void Pointers (cont'd)

So how does this help us make generic classes?

- We use `void*` as the data type!
- And type cast it when using it .

**Let us now look
at workshop 6
656question**

