Introduction to Programming with C++

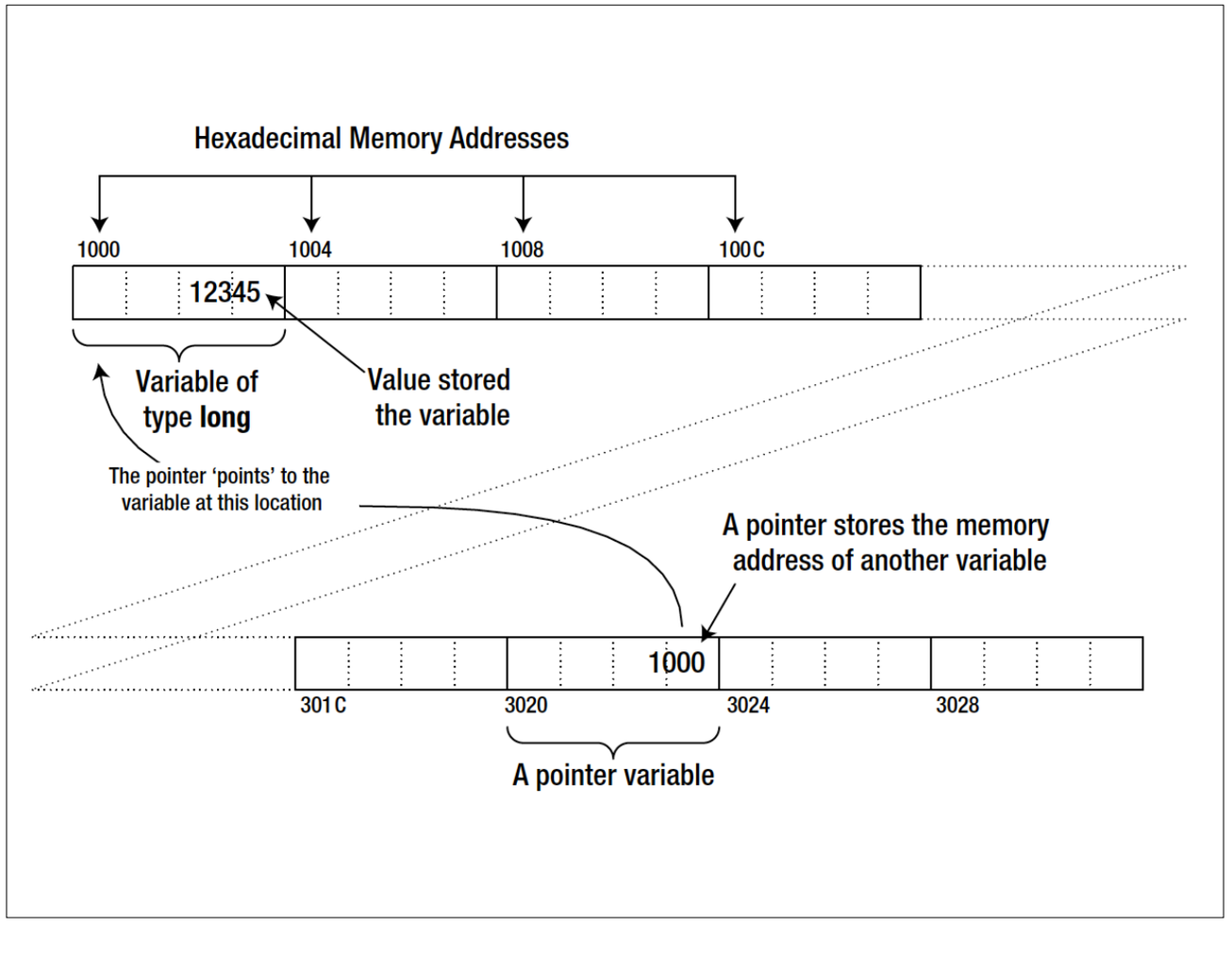
Pointers and References

Objective

- In this chapter, you'll learn:
  - What pointers are and how they are defined
  - How to obtain the address of a variable
  - How to create memory for new variables while your program is executing
  - How to release memory that you've allocated dynamically
  - The difference between raw pointers and smart pointers
  - How to create and use smart pointers
  - How you can convert from one type of pointer to another
  - What a reference is and how it differs from a pointer

What Is a Pointer?

- A pointer is a variable that can store an address.
- The address stored can be the address of a variable or the address of a function.
- The figure in the next slide shows how a pointer gets its name
- It “points to” a location in memory where something (a variable or a function) is stored.
- However, a pointer needs to record more than just a memory address to be useful.
- A pointer must store what is at the address, as well as where it is.
- An integer has a different representation from a floating-point value, and the number of bytes occupied by an item of data depends on what it is.
- To use a data item stored at the address contained in a pointer, you need to know the type of the data.


Hexadecimal Memory Addresses
1000
1004
1008
100C
12345
Variable of type long
Value stored the variable
The pointer 'points' to the variable at this location
A pointer stores the memory address of another variable
1000
301C
3020
3024
3028
A pointer variable

What Is a Pointer?

- Thus a pointer isn't just a pointer to an address
- It's a pointer to a particular type of data item at that address.
- The definition of a pointer is similar to that of an ordinary variable except that the type name has an asterisk following it to indicate that it's a pointer and not a variable of that type.
- Here's how you define a pointer called pnumber that can store the address of a variable of type long:

```
long* pnumber {}; // A pointer to type long
```

- The type of pnumber is "pointer to long", which is written as `long*`.
- This pointer can only store an address of a variable of type long.

What Is a Pointer?

- An attempt to store the address of a variable that is other than type long will not compile.
- Because the initializer list is empty, the statement initializes pnumber with the pointer equivalent of zero, which is an address that doesn't point to anything.
- The equivalent of zero for a pointer is written as nullptr, and you could specify this explicitly as the initial value:

```
long* pnumber {nullptr};
```

- You are not obliged to initialize a pointer when you define it but it's reckless not to.
- Uninitialized pointers are more dangerous than ordinary variables that aren't initialized
- Note Always initialize a pointer when you define it.

What Is a Pointer?

- You can position the asterisk adjacent to the variable name, like this:

```
long *pnumber {nullptr};
```

- This defines precisely the same variable as before.
- The compiler accepts either notation.
- The former is perhaps more common because it expresses the type, “pointer to long,” more clearly.
- However, there’s potential for confusion if you mix definitions of ordinary variables and pointers in the same statement.
- Try to guess what this statement does:

```
long* pnumber {}, number {};
```

- This defines pnumber of type “pointer to long” initialized with nullptr and number as type long initialized with 0L.
- The notation that juxtaposes the asterisk and the type name makes this less than clear.
- It’s a little clearer if you define the two variables in this form:

```
long *pnumber {}, number {};
```

- This is less confusing because the asterisk is now clearly associated with the variable pnumber.
- However, the real solution is to avoid the problem in the first place.
- It’s always better to define pointers and ordinary variables in separate statements:

```
long number {}; // Variable of type long
long* pnumber {}; // Variable of type 'pointer to long'
```

The Address-Of Operator

- The address-of operator, &, is a unary operator that obtains the address of a variable.
- You could define a variable, number, and a pointer, pnumber, initialized with the address of number with these statements:

```
long number {12345L};
long* pnumber {&number};
```

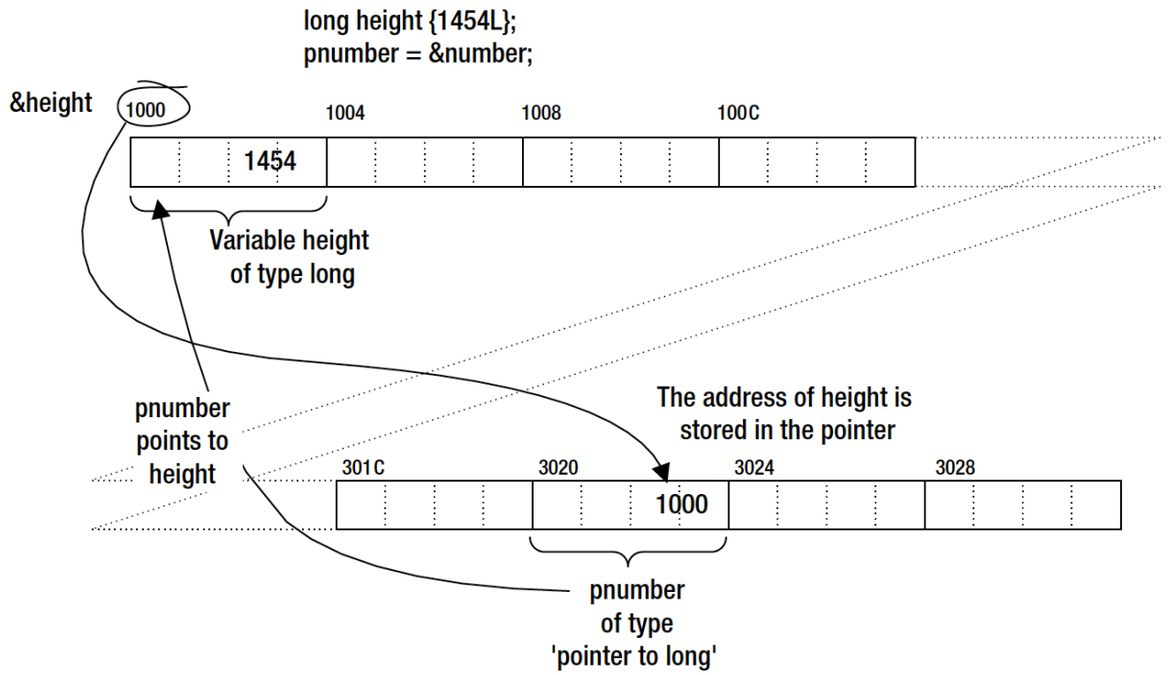
- &number produces the address of number so pnumber has this address as its initial value.
- pnumber can store the address of any variable of type long so you can write the following assignment:

```
long height {1454L}; // Stores the height of a building
pnumber = &prime; // Store the address of height in pnumber
```

- The result of the statement is that pnumber contains the address of height.
- The effect is illustrated with the Figure in the next slide.

- The & operator can be applied to a variable of any type, but you can only store the address in a pointer of the appropriate type.
- If you want to store the address of a double variable for example, the pointer must have been declared as type double*, which is “pointer to double”.

The Address-Of Operator



The Indirection Operator

- Accessing the data in the memory location to which the pointer points is done using the ***indirection operator***.
- Applying the indirection operator, *, to a pointer accesses the contents of the memory location to which it points.
- The name “indirection operator” stems from the fact that the data is accessed “indirectly”.
- The operator is sometimes called the ***dereference*** operator
- The process of accessing the data in the memory location pointed to by a pointer is termed dereferencing the pointer.
- To access the data at the address contained in the pointer, pnumber, you use the expression *pnumber.

See Ex6_01.cpp

Why Use Pointers?

- You can use pointer notation to operate on data stored in an array, which may execute faster than if you use array notation.
- When you define your own functions , you'll see that pointers are used extensively to enable a function to access large blocks of data, such as arrays, that are defined outside the function.
- You can allocate memory for new variables dynamically — that is, during program execution.
- This allows a program to adjust its use of memory depending on the input.
- You can create new variables while your program is
- executing, as and when you need them.
- When you allocate new memory, the memory is identified by its address so you need a pointer to record it.
- Pointers are fundamental to enabling polymorphism to work.

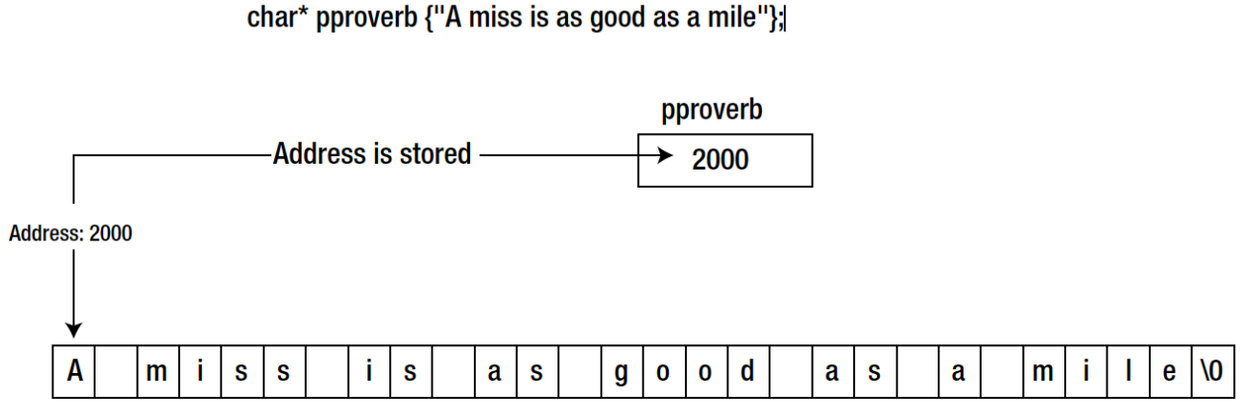
Pointers to Type char

- A variable of type “pointer to char” has the interesting property that it can be initialized with a string literal.
- For example, you can declare and initialize such a pointer with this statement:

```
char* pproverb {"A miss is as good as a mile."};
```

- This looks very similar to initializing a char array with a string literal, and indeed it is.
- The statement creates a null-terminated string literal (actually, an array of elements of type const char) from the character string between the quotes and stores the address of the first character in pproverb.

Pointer of type `char*`



The string literal is stored as a null-terminated string.

Pointer of type `char*`

- The type of the string literal is const, but the type of the pointer is not.
- The statement doesn't create a modifiable copy of the string literal; it merely stores the address of the first character.
- This means that if you attempt to modify the string, there will be trouble.
- Look at this statement, which tries to change the first character of the string to 'X':

```
*pproverb = 'X';
```

- Some compilers won't complain, because they see nothing wrong. The pointer, pproverb, wasn't declared as const, so the compiler is happy.
- With other compilers you get a warning that there is a deprecated conversion from type const char* to type char*.
- In some environments you'll get an error when you run the program, resulting in a program crash.
- In other environments the statement does nothing, which presumably is not what was required or expected.

You might wonder, with good reason, why the compiler allowed you to assign a const value to a non-const type in the first place, particularly when it causes these problems. The reason is that string literals only became constants with the release of the first C++ standard, and there's a great deal of legacy code that relies on the "incorrect" assignment.

Constant Pointers & Pointers to Constants

- You can distinguish three situations that arise using const when applied to pointers and the things to which they point
  - A pointer to a constant
    - You can’t modify what’s pointed to, but you can set the pointer to point to something else

```
const char* pstring {"Some long
string"};
const int value {20};
const int* pvalue {&value};
```

Constant Pointers & Pointers to Constants

- A constant pointer
  - The address stored in the pointer can't be changed.
  - A constant pointer can only ever point to the address that it's initialized with.
  - However, the contents of that address aren't constant and can be changed.

```
int data {20};
int* const pdata {&data};
```

Constant Pointers & Pointers to Constants

- A constant pointer to a constant
  - Both the address stored in the pointer and the item pointed to are constant
  - so neither can be changed.

```
const int value {20};
const int* const pvalue {&value};
```

Pointers and Arrays

- There are many situations in which you can use an array name as though it were a pointer.
- An array name by itself can behave like a pointer when it's used in an output statement.
- If you try to output an array by just using its name, unless it's a char array you'll get is the hexadecimal address of the array.
- Because an array name can be interpreted as an address, you can use an array name to initialize a pointer:

```
double values[10];
double* pvalue {values};
```

- This will store the address of the values array in the pointer pvalue.
- Although an array name represents an address, it is not a pointer.
- You can modify the address stored in a pointer, whereas the address that an array name represents is fixed.

Pointer Arithmetic

- Arithmetic with pointers works in a special way.
- Suppose you add 1 to a pointer with a statement such as this:

```
++pvalue;
```

- This apparently increments the pointer by 1.
- Exactly how you increment the pointer by 1 doesn't matter.
- You could use an assignment or the += operator to obtain the same effect so the result would be exactly the same with this statement:

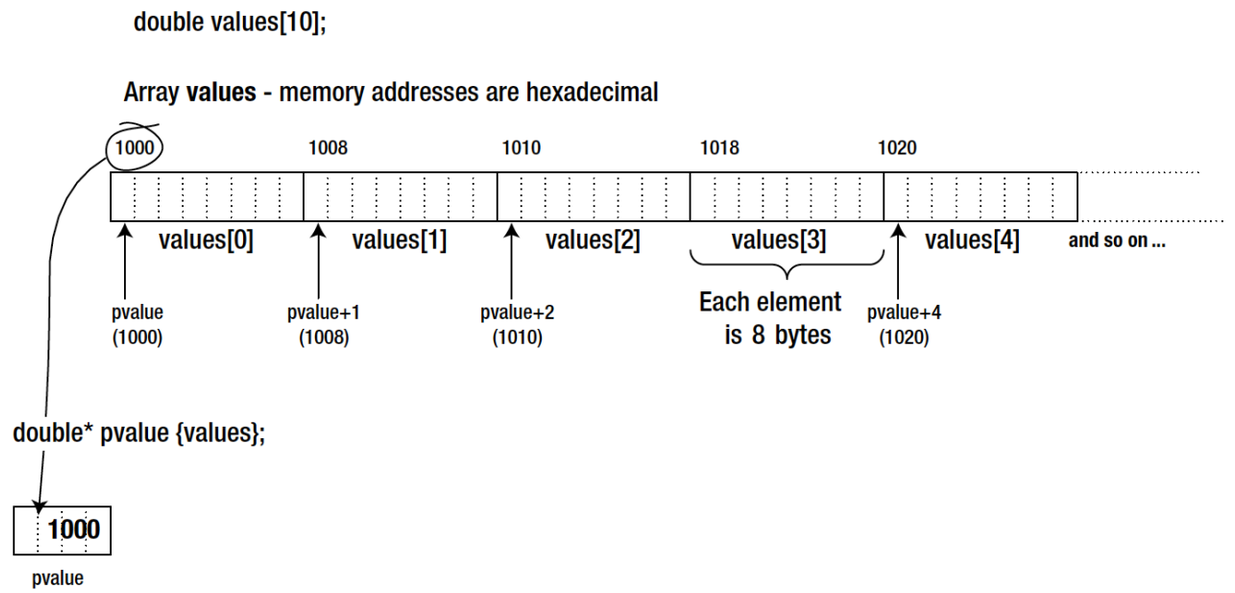
```
pvalue += 1;
```

- The address stored in the pointer won't be incremented by 1 in the normal arithmetic sense.

Pointer Arithmetic

- Pointer arithmetic implicitly assumes that the pointer points to an array.
- Incrementing a pointer by 1 means incrementing it by one element of the type to which it points.
- The compiler knows the number of bytes required to store the data item to which the pointer points.
- Adding 1 to the pointer increments the address by that number of bytes.
- In other words, adding 1 to a pointer increments the pointer so that it points to the next element in the array.
- For example, if pvalue is “pointer to double” and type double is 8 bytes, then the address in pvalue will be incremented by 8.

Incrementing a pointer



Using Pointer Notation with an Array Name

- You can use an array name as though it was a pointer for addressing the array elements.
- Suppose you define this array:

```
long data[5] {};
```

- You can refer to the element data[3] using pointer notation as

```
*(data + 3).
```

- The array name by itself refers to the address of the beginning of the array
- So an expression such as `data+2` produces the address of the element two elements along from the first.

See Ex6_04.cpp

Dynamic Memory Allocation

- All of the code you've written up to now allocates space for data at compile time.
- You specify the variables and the arrays that you need in the code, and that's what will be allocated when the program starts, whether you need it or not.
- Working with a fixed set of variables in a program can be very restrictive, and it's often wasteful.
- Dynamic memory allocation is allocating the memory you need to store the data you're working with at runtime,
- rather than having the amount of memory predefined when the program is compiled.

Dynamic Memory Allocation

- You can change the amount of memory your program has dedicated to it as execution progresses.
- By definition, dynamically allocated variables can't be defined at compile time so they can't be named in your source program.
- When you allocate memory dynamically, the space that is made available is identified by its address.
- The obvious and only place to store this address is in a pointer.
- With the power of pointers and the dynamic memory management tools in C++, writing this kind of flexibility
- into your programs is quick and easy.
- You can add memory to your application when it's needed, then release the memory you have acquired when you are done with it.
- Thus the amount of memory dedicated to an application can increase and decrease as execution progresses.

Using the `new` Operator

- Suppose you need space for a variable of type double.
- You can define a pointer of type double* and then request that the memory is allocated at execution time. Here's one way to do this:

```
double* pvalue {}; // Pointer initialized with nullptr
pvalue = new double; // Request memory for a double variable
```

- This is a good moment to recall that all pointers should be initialized.
- Using memory dynamically typically involves having a lot of pointers floating around, and it's important that they not contain spurious values.

Using the `new` Operator

- You should always ensure that a pointer contains nullptr if it doesn't contain a legal address.
- The new operator in the second line of the code returns the address of the memory in the free store allocated to a double variable, and this is stored in pvalue.
- You can use this pointer to reference the variable in the free store using the indirection operator as you've seen.

```
*pvalue = 3.14;
```

- You can initialize a variable that you create in the free store.

```
pvalue = new double {3.14};
```

Using the `delete` Operator

- You can also create and initialize in the same line

```
double* pvalue {new double {3.14}};
```

- When you no longer need a dynamically allocated variable, you free the memory that it occupies using the

```
delete operator:
delete pvalue; // Release memory pointed to by pvalue
```

- This ensures that the memory can be used subsequently by another variable.
- If you don't use delete, and you store a different address in pvalue, it will be impossible to free up the original memory because access to the address will have been lost.

Using the `delete` Operator

- Note that the delete operator frees the memory but does not change the pointer.
- After the previous statement has executed, pvalue still contains the address of the memory that was
- allocated, but the memory is now free and may be allocated immediately to something else — possibly by another program.
- The pointer now contains a spurious address so you should always reset a pointer when you release the memory to which it points, like this:

```
delete pvalue; // Release memory pointed to by pvalue
pvalue = nullptr; // Reset the pointer
```

Dynamic Allocation of Arrays

- Allocating memory for an array at runtime is straightforward.

```
double* pdata {new double[20]};
```

- This allocates space for an array of 100 values of type double and stores its address in pdata.
- To remove the array from the free store when you are done with it, you use the delete operator, but a little differently:

```
delete [] pdata; // Release array pointed to by pdata
```

- The square brackets are important because they indicate that you're deleting an array.
- When removing arrays from the free store, you must include the square brackets or the results will be unpredictable.
- Note that you don't specify any dimensions, simply [].
- Of course, you should also reset the pointer, now that it no longer points to memory that you own:

```
pdata = nullptr; // Reset the pointer
```

See Ex6_05.cpp

Summary

- The essential points you have learned in this chapter are:
  - A pointer is a variable that contains an address.
  - A basic pointer is referred to as a raw pointer.
  - You obtain the address of a variable using the address-of operator, &.
  - To refer to the value pointed to by a pointer, you use the indirection operator, *.
  - This is also called the dereference operator.
  - The new operator allocates a block of memory in the free store and returns the address of the memory allocated.
  - You use the delete operator to release a block of memory that you've allocated previously
  - using the new operator.

Reading Assignment

1. The Stack and the Heap (Page 167)
2. C Program memory layout (Additional Resource)
3. Hazards of Dynamic Memory Allocation(Page 171-172)
4. Raw and Smart Pointers (Page 173-179)
5. Understanding References(Page 180-182)

Exercise

1. Write a program that declares and initializes an array with the first 50 even numbers. Output the numbers from the array ten to a line using pointer notation, and then output them in reverse order also using pointer notation.
2. Write a program that reads an array size from the keyboard and dynamically allocates an array of that size to hold floating-point values. Using pointer notation, initialize all the elements of the array so that the value of the element at index position n is 1.0/(n+1)2. Calculate the sum of the elements using pointer notation, multiply the sum by 6, and output the square root of that result.
3. Repeat the calculation in Exercise 2 but using a vector<> container allocated in the free store. Test the program with more than 100,000 elements. Do you notice anything interesting about the result?