CSE2050
Programming in a Second Language (C++)
Today’s lecture

- Using the word `const`
- Dynamic memory allocation
- Function that return a pointer
- Handling allocation errors
- Pointers to functions
Chapter 9 Pointers

const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45, 12.85, 14.97, 10.35, 18.89 };

In this code, 
\textit{payRates} is an array of \textit{const double} s. This means that each element in the array is a \textit{const double}, and the compiler will not allow us to write code that changes the array's contents. If we want to pass the \textit{payRates} array into a pointer parameter, the parameter must be declared as a pointer to \textit{const double}. The following function shows such an example:

\begin{verbatim}
void displayPayRates(const double *rates, int size)
{ // Set numeric output formatting.
  cout << setprecision(2) << fixed << showpoint;
  // Display all the pay rates.
  for (int count = 0; count < size; count++)
    cout << "Pay rate for employee " << (count + 1)
         << " is \$" << *(rates + count) << \endl;
}
\end{verbatim}

In the function header, notice that the \textit{rates} parameter is defined as a pointer to \textit{const double}. It should be noted that the word \textit{const} is applied to the thing that \textit{rates} points to, not \textit{rates} itself. This is illustrated in Figure 9-8.

Because \textit{rates} is a pointer to a \textit{const}, the compiler will not allow us to write code that changes the thing that \textit{rates} points to.

Passing a Nonconstant Argument into a Pointer to a Constant

Although a constant's address can be passed only to a pointer to \textit{const}, a pointer to \textit{const} can also receive the address of a nonconstant item. For example, look at Program 9-13.

Figure 9-8

\textbf{Points to constant variables}

The asterisk indicates that \textit{rates} is a pointer.

\begin{verbatim}
const double *rates
\end{verbatim}

This is what \textit{rates} points to.
Pointers to constant variables

const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45, 
12.85, 14.97, 
10.35, 18.89 };
Pointers to constant variables

```c
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                                12.85, 14.97,
                                10.35, 18.89 };

void displayPayRates(const double *rates, int size)
{
    // Set numeric output formatting.
    cout << setprecision(2) << fixed << showpoint;

    // Display all the pay rates.
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1)
             << " is "$ << *(rates + count) << endl;
    }
}
```
Constant points to non-constant variables


code

```c
int value = 22;
int * const ptr = &value;
```

**Figure 9-9**

```
int * const ptr
```

This is what `ptr` points to.
Constant points to non-constant variables

Constant pointers must be initialized.

```c
void setToZero(int * const ptr)
{
    ptr = 0; // ERROR!!
}
```

When used as a function parameter, pointer is initialized with the address that is passed to the function.
This function will compile just fine.

```c
void setToZero(int * const ptr)
{
    *ptr = 0;
}
```

When used as a function parameter, pointer is initialized with the address that is passed to the function.
The word `const` appears before `int`, indicating that `ptr` points to a `const int`, and it appears after the asterisk, indicating that `ptr` is a constant pointer.
We cannot write code that makes \texttt{ptr} point to anything else.

Because \texttt{ptr} is a pointer to \texttt{const}, we cannot use it to change the contents of value.
iptr = new int;
*iptr = 25;
cout << *iptr;
cin >> *iptr;
total += *iptr;

**iptr** will contain the address of the newly allocated memory.
Dynamic memory allocation: arrays

```cpp
iptr = new int[100];
if (iptr == NULL)
{
    cout << "Error allocating memory!\n";
    return;
}
```

`iptr` will contain the address of the newly allocated memory.
Although the statements above illustrate the use of the `new` operator, there's little purpose in dynamically allocating a single variable. A more practical use of the `new` operator is to dynamically create an array. Here is an example of how a 100-element array of integers may be allocated:

```
iptr = new int[100];
```

Once the array is created, the pointer may be used with subscript notation to access it. For instance, the following loop could be used to store the value 1 in each element:

```
for (int count = 0; count < 100; count++)
    iptr[count] = 1;
```

But what if there isn't enough free memory to accommodate the request? What if the program asks for a chunk large enough to hold a 100,000-element array of `float`s, and that much memory isn't available? When memory cannot be dynamically allocated, C++ throws an exception and terminates the program. Throwing an exception means the program signals that an error has occurred.

Programs created with older C++ compilers behave differently when memory cannot be dynamically allocated. Under older compilers, the `new` operator returns the address 0, or `NULL` when it fails to allocate the requested amount of memory. (`NULL` is a named constant, defined in the `iostream` file, that stands for address 0.) A program created with an older compiler should always check to see if the `new` operator returns `NULL`, as shown in the following code:

```
iptr = new int[100];
if (iptr == NULL) {
    cout << "Error allocating memory!
";
    return;
}
```

---

**Figure 9-11**

Dynamic memory allocation: arrays

- `iptr` variable
- Pool of unused memory
  - This chunk of memory starts at address 0xA652
Clearing up dynamically allocated memory

```cpp
delete iptr;

delete [] iptr;
```
Clearing up dynamically allocated memory:
good practice to set pointer to 0 after delete

```c++
delete [] iptr;
iptr = 0;
```

- It prevents code from inadvertently using the pointer to access the area of memory that was freed.
- It prevents errors from occurring if `delete` is accidentally called on the pointer again.
- The `delete` operator has no effect on a null pointer.
Functions that return a pointer

cchar *getName()
{
  char *name;

  name = new char[81];
  cout << "Enter your name: ";
  cin.getline(name, 81);
  return name;
}
Functions that return a pointer

```c
char *findNull(char *str)
{
    char *ptr = str;

    while (*ptr != '\0')
        ptr++;

    return ptr;
}
```
Functions that return a pointer: what is wrong?

```cpp
char *getName()
{
    char name[81];

    cout << "Enter your name: ";
    cin.getline(name, 81);

    return name;
}
```
What if memory cannot be allocated by OS?

» Old compilers: test pointer for NULL value

```cpp
iptr = new int[100];
if (iptr == NULL)
{
    cout << "Error allocating memory!\n";
    return;
}
```
What if memory cannot be allocated by OS?

» Old compilers: test pointer for NULL value

```cpp
iptr = new int[100];
if (!iptr)
{
    cout << "Error allocating memory!\n";
    return;
}
```

This style is quite common
What if memory cannot be allocated by OS?

» New compilers: throws a `bad_alloc` exception and terminates program

» You can:
  - Mask the exception and handle the issue in the old style.
  - Use a try/catch construct to handle the exception
p = new (nothrow) int[i];
if (!p)
    cout << "Error: memory could not be allocated";
else
{
    // everything is fine, do something with array
    (...) 
    delete[] p;
}
The try/catch method (bad_alloc)

double *ptr;

try {
    ptr = new double [10000];
} catch (bad_alloc) {
    cout << "insufficient memory. \n";
}

// Use the array that has been allocated (…)

Pointers to functions

C++ allows operations with pointers to functions.

Typical use: passing a function as an argument to another function.
void my_int_func(int x)
{
    printf( "%d\n", x );
}

int main()
{
    void (*foo)(int);    // Declare a pointer to a function
    foo = &my_int_func;  // Get the address of the function

    foo( 2 );            // Call the function - style 1
    (*foo)( 2 );         // Call the function - style 2

    return 0;
}