More on Functions

Function Overloading Example

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

void Swap(char&, char&);
void Swap(int&, int&);
void Swap(double&, double&);

int main()
{
   char c1 = 'x', c2 = 'y';
   int i1 = 11, i2 = 22;
   double d1 = 11.11, d2 = 22.22;

   cout << "Before swapping:" << endl;
   cout << "\nc1 = " << c1 << "; c2 = " << c2 << endl;
   cout << "\ni1 = " << i1 << "; i2 = " << i2 << endl;
   cout << "\nd1 = " << d1 << "; d2 = " << d2 << endl;

   Swap(c1, c2);
   Swap(i1, i2);
   Swap(d1, d2);

   cout << "After swapping:" << endl;
   cout << "\nc1 = " << c1 << "; c2 = " << c2 << endl;
   cout << "\ni1 = " << i1 << "; i2 = " << i2 << endl;
   cout << "\nd1 = " << d1 << "; d2 = " << d2 << endl;

   return EXIT_SUCCESS;
}
```

(continued on next slide)

More on Functions

Function Overloading Example

(continued from previous slide)

```
void Swap(char& first, char& second)
{
   char temp = first;
   first = second;   second = temp;
}

void Swap(int& first, int& second)
{
   int temp = first;
   first = second;   second = temp;
}

void Swap(double& first, double& second)
{
   double temp = first;
   first = second;   second = temp;
}
```

More on Functions — Function Overloading

- If two or more different functions are given the same name, that name is said to be ***overloaded***
  - Often the set of functions sharing that same name are collectively considered to be one single function – one that has been overloaded
- The name of a function can be overloaded, provided no two definitions of the function have the same ***signature***
  - The signature of a function is a ***list of the types of its parameters***, including any `const` and reference or pointer indicators (`&` or `*`)
  - Examples: (for the overloaded `Swap` function just discussed)

    `(char&, char&)` `(int&, int&)` `(double&, double&)`
  - Function signatures are important
    - Compiler essentially considers a function's signature to be part of its name
    - Allow compiler to distinguish ("disambiguate") calls to different functions with the same name – make function overloading possible
    - *Caveat:* compiler may not be able to "disambiguate" certain signature difference – e.g.: `Calc(int)` vs `Calc(int&)` in call `Calc(intVal);`

More on Functions — Function Overloading

- ***Repeat:*** The name of a function can be overloaded, provided no two definitions of the function have the same ***signature***
  - In general, different function signatures result if...
    - the functions differ in **NAME** (non-factor in function overloading ← same name)

      e.g.: `void Print(int);` vs `void Display(int, int);`
    - *and/or* the parameters of the functions differ in **NUMBER**

      e.g.: `void Print(int);` vs `void Print(int, int);`
    - *and/or* the parameters of the functions differ in **TYPE**

      e.g.: `void Print(int);` vs `void Print(double);`
    - *and/or* the types of the parameters of the functions differ in **ORDER**

      e.g.: `void Print(int, char);` vs `void Print(char, int);`
  - **RETURN TYPE** has ***no*** role in determining a function's signature
    - Two functions with *identical lists of parameter types* but *different return types* *cannot* be overloaded (*i.e.*, they cannot be made to share a common name)

More on Functions

Function Signatures Quick Quiz

Which of these functions have the same signature?

- ☐ `int calc(int a, int b);`
- ☐ `int calc(int c, int d);`
- ☐ `int calc(int a, int& b);`
- ☐ `int calc(int a, const int b);`
- ☐ `int calc(int a, const int& b);`
- ☐ `int calc(int a, int* b);`
- ☐ `int calc(int a, const int* b);`
- ☐ `int calc(int a, int*& b);`
- ☐ `void calc(int a, int b);`
- ☐ `int calc(int a, double b);`
- ☐ `int calc(double b, int a);`
- ☐ `int calc(int a, int b, int c);`

More on Functions

Function Overloading

- Names should be overloaded only when it is appropriate
  - Different functions that perform the same operation (*e.g.*, summation, swapping, find the minimum or maximum) on different data types are prime candidates for overloading
  - Giving operations that have nothing to do with each other the same name simply because the language allows you to do so…
    - is an abuse of the overloading mechanism as well as bad programming style
- The basic C/C++ operators `+`, `-`, `*`, and `/` are all overloaded
  - In the expression `(2.0/5.0)` the C/C++ compiler uses the real division operation, which produces the value 0.4
  - In the expression `(2/5)` the C/C++ compiler uses the integer division operation, which produces the value 0
  - Many of the other operators, including `<<`, `>>`, `=`, `=`, `+=`, `-=`, `*=`, `/=`, `<`, `>`, `==`, `<=`, `>=`, and `!=` have also been overloaded

More on Functions

Default Arguments

- In some situations...
  - we can declare and define a single function with ***default argument(s)***...
  - instead of providing two or more different function declarations and definitions (and giving them the same function name)
- A function with default arguments...
  - can be called without all of its arguments specified in the call
  - (when arguments are unspecified in a call, the corresponding parameters receive the default argument values)

More on Functions

Default Arguments

```
#include <iostream>
#include <cstdlib>
using namespace std;
int Sum(int a, int b, int c = 0, int d = 0);
int main()
{
   int a = 2, b = 5, c = 12, d = 34;
   cout << "(a+b) = " << Sum(a, b) << endl;
   cout << "(a+b+c) = " << Sum(a, b, c) << endl;
   cout << "(a+b+c+d) = " << Sum(a, b, c, d) << endl;
   return EXIT_SUCCESS;
}

int Sum(int a, int b, int c, int d)
{
   return (a + b + c + d);
}
```

Caveat:
values for defaults arguments can be specified either in the function's prototype or in the function's header, but **not** in both places

More on Functions
Default Arguments

```
#include <iostream>
#include <cstdlib>
using namespace std;
int BoxVol(int L = 1, int W = 1, int H = 1);
int main()
{
   cout << "Default box volume is: " << BoxVol() << endl;
   cout << "Volume of box with L=10, W=1, H=1 is: "
        << BoxVol(10) << endl;
   cout << "Volume of box with L=10, W=5, H=1 is: "
        << BoxVol(10, 5) << endl;
   cout << "Volume of box with L=10, W=5, H=2 is: "
        << BoxVol(10, 5, 2) << endl;
   return EXIT_SUCCESS;
}
int BoxVol(int length, int width, int height)
{
   return (length * width * height);
}
```


More on Functions
Default Arguments

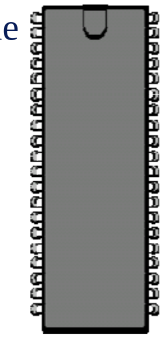
Rules for Functions with Default Argument(s)

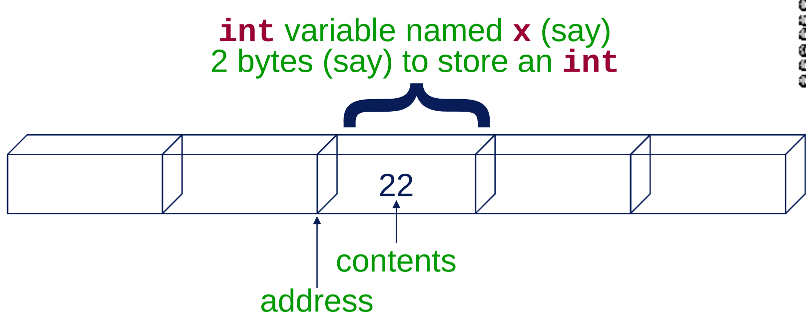
- Default arguments must be the ***rightmost*** (***trailing***) arguments in function's parameter list
  - ✓ `int Sum(int a, int b, int c=0, int d=0);`
  - ✗ `int Sum(int a, int b=0, int c, int d=0);`
  - ✗ `int Sum(int a=0, int b=0, int c, int d);`
- Default values are specified in a function's ***declaration*** but are ***not*** repeated in the ***definition***

Memory Concepts

4 important attributes associated with every variable

- ***name*** – identifier for referencing variable directly
- ***value stored*** – variable's content
- ***address*** – address of *first* memory location allocated for the variable
- ***data type*** – determines # of memory locations used





Memory Concepts

To determine the address of a variable:

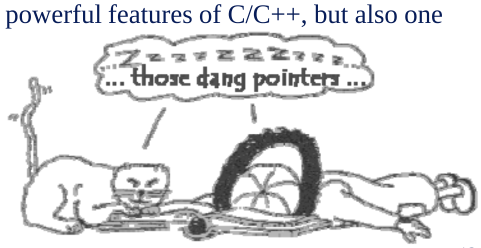
- attach ***address of*** operator (**&**) immediately in front of the variable name
- *e.g.*: **`&years`** (NOTE: no space between **&** and the variable name)

To store addresses of variables:

- special variables must be declared for the purpose
- such variables are called ***pointer variables*** or simply ***pointers***
- (pointers are simply special variables for storing memory addresses)
- (pointers represent one of the most powerful features of C/C++, but also one of the most difficult to master)

Two popular uses of pointers:

- simulate call by reference (in C)
- implement linked data structures (linked lists, trees, *etc.*)



Memory Concepts

With the address of a variable stored in a pointer...

- we now have two ways to reference the value stored in the memory location(s) associated with the variable...
- either ***directly*** by using the variable's name...
- or ***indirectly*** by using the pointer – through a process called ***indirection*** or ***dereferencing*** (will see how in a few)

... oh well, that's cute ...but why ?!

Memory Concepts

Pointers, like any other variables, must be declared before they can be used

Example: **`int *countPtr = 0, count;`**

- declares **`countPtr`** to be of type **`int *`** (*i.e.*, pointer to **`int`**) and initializes it to **`0`** (more on the "zero" address to come)
  - is read "**`countPtr`** is a pointer to **`int`**"
  - or "**`countPtr`** points to an object of type **`int`**"
- C/C++ treats **`int *`**, **`double *`**, **`char *`**, *etc.* as ***different types***
  - "pointer" thus represents a *family of types* (not just one type)
- when **`*`** is used in this manner in *declaration statements*, it indicates that the variable being declared is a pointer
  - same symbol is also used to dereference pointers in *executable statements* (*i.e.*, used as the ***indirection*** or ***dereferencing operator*** – to be discussed)
    ➔ many beginners are confused by this (so beware)

Memory Concepts
Pointer Basics

Example (cont'd): `int *countPtr = 0, count;`

- also declares `count` to be an `int`
  - NOT a pointer to `int`
  - `*` applies only to `countPtr` in the declaration
  - `*` does NOT distribute to all variable names in declaration statements
  - each pointer must be declared with `*` prefixed to its name
- good practice to include suffix `Ptr` in pointer variable names
  - helps reduce the potential for mix-ups between a normal variable and a pointer variable

Memory Concepts
Pointer Basics

Pointers can be declared to point to...

- objects of any data type – including programmer-defined data type
- (recall that a "pointer to a type" is itself a separate and different type)

Pointers should be ***initialized*** when they are declared...

- otherwise a memory location that is not legally accessible (but quite likely pointed to by the "garbage" address contained in an uninitialized pointer) may accidentally be accessed

A pointer may be initialized to...

- either **0** (the ***only integer value*** that can be assigned directly to a pointer and it can be assigned to a pointer to ***any type***)...
- or the ***address*** of a variable of the ***same type*** pointed to by the pointer

Memory Concepts

Pointer Basics

- A pointer with a value **0** (the ***null address***)...
  - is called the ***null pointer***
  - is intended to mean that the pointer points to nothing
    - doesn't contain the address of any legally accessible memory location
  - cannot be accessed because the memory location with the null address is reserved
    - any attempt to do so will result in the system terminating the program
- Initializing or assigning a pointer with the null address when there is *no other appropriate address* for the pointer (perhaps for the time being) is a way of preventing the pointer from being improperly used
  - (it is considered better to terminate a program when a pointer has been improperly used than to let the program continue to run)
  - (there are some that are against using the null address this way)

Memory Concepts

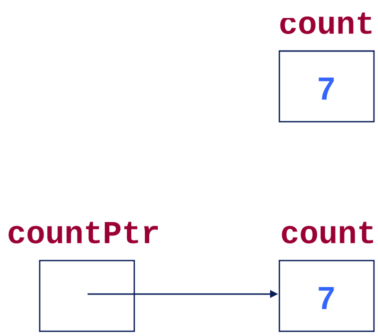
Pointer Basics

Example (cont'd and extended):

```
int *countPtr = 0, count;
...
count = 7;
countPtr = &count;
...
```

How are `count` and `countPtr` related?

Memory Concepts



count directly references a variable whose value is 7

countPtr indirectly references (or points to) a variable whose value is 7

Memory Concepts

Pointer Basics

With the address of some "legally accessible" memory location properly stored in a pointer, the pointer can then be used to indirectly access that memory location

- Example:

```
...
int y = 5, x;
int *yPtr = &y;
cout << y << endl;
*yPtr = 10;
x = *yPtr;
...
```

`yPtr` is a "**pointer to `int`**" initialized to the address of `int` variable `y`

"**memory location whose address is stored in `yPtr`**" is assigned a value of 10

"**<u>value that is stored in</u> memory location whose address is stored in `yPtr`**" is assigned to `x`

- When `*` is used in ***executable statements*** as in the last two statements shown above, it has the meaning of an operator operating on the pointer
  - ***indirection*** or ***dereferencing operator*** (a *unary* operator) – not to be confused with the multiplication operator (a *binary* operator)

Memory Concepts

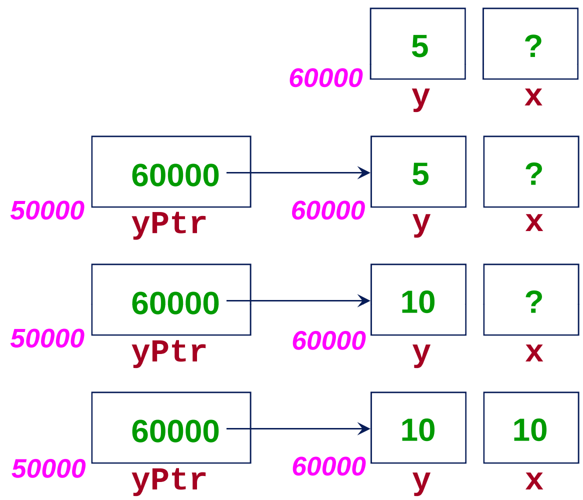
Pointer Basics

```
int y = 5, x;

int *yPtr = &y;

*yPtr = 10;

x = *yPtr;
```



Memory Concepts

Pointer Basics

When used with standard input and output stream objects (also with file input and output stream objects:

```
...
int y;
int *yPtr = &y;
cout >> "Enter value for y: ";
cin >> *yPtr;
cout << "y now has a value of ";
cout << *yPtr
     << endl;
...
```

`yPtr` is a "**pointer to `int`**" initialized to the address of **`int`** variable **`y`**

"**memory location whose address is stored in `yPtr`**" is assigned a value read from standard input

"**<u>value that is stored in</u> memory location whose address is stored in `yPtr`**" is written to standard output

Memory Concepts

Pointer Basics

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

int main()
{
   int x, *xPtr = 0;

   x = 7;
   xPtr = &x;
   cout << "Address of x is " << &x << endl;
   cout << "Value of xPtr is " << xPtr << endl;
   cout << "Value of x is " << x << endl;
   cout << "Value of *xPtr is " << *xPtr << endl;

   return EXIT_SUCCESS;
}
```

```
Address of x is 0x0f860ffe
Value of xPtr is 0x0f860ffe
Value of x is 7
Value of *xPtr is 7
```


Memory Concepts

Pointer Basics

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

int main()
{
   int x = 42, y = 80;
   int *xPtr = &x,
       *yPtr = &y;

   *xPtr = *yPtr;

   cout << "Address of x is " << &x << endl;
   cout << "Address of y is " << &y << endl;
   cout << "Value of xPtr is " << xPtr << endl;
   cout << "Value of yPtr is " << yPtr << endl;
   cout << "Value of x is " << x << endl;
   cout << "Value of y is " << y << endl;

   return EXIT_SUCCESS;
}
```

```
Address of x is 0x0012FF7C
Address of y is 0x0012FF78
Value of xPtr is 0x0012FF7C
Value of yPtr is 0x0012FF78
Value of x is 80
Value of y is 80
```

Memory Concepts

Pointer Basics

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

int main()
{
   int x = 42, y = 80;
   int *xPtr = &x,
       *yPtr = &y;

   xPtr = yPtr;

   cout << "Address of x is " << &x << endl;
   cout << "Address of y is " << &y << endl;
   cout << "Value of xPtr is " << xPtr << endl;
   cout << "Value of yPtr is " << yPtr << endl;
   cout << "Value of x is " << x << endl;
   cout << "Value of y is " << y << endl;

   return EXIT_SUCCESS;
}
```

```
Address of x is 0x0012FF7C
Address of y is 0x0012FF78
Value of xPtr is 0x0012FF78
Value of yPtr is 0x0012FF78
Value of x is 42
Value of y is 80
```


Memory Concepts

Pointer Basics

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

int main()
{
   int x = 42, y = 80;
   int *xPtr = &x,
       *yPtr = &y;

   xPtr = yPtr;
   *xPtr = 52;
   *yPtr = 62;

   cout << "Address of x is " << &x << endl;
   cout << "Address of y is " << &y << endl;
   cout << "Value of xPtr is " << xPtr << endl;
   cout << "Value of yPtr is " << yPtr << endl;
   cout << "Value of x is " << x << endl;
   cout << "Value of y is " << y << endl;

   return EXIT_SUCCESS;
}
```

```
Address of x is 0x0012FF7C
Address of y is 0x0012FF78
Value of xPtr is 0x0012FF78
Value of yPtr is 0x0012FF78
Value of x is 42
Value of y is 62
```

Memory Concepts

Pointer Basics

we've seen how C++ allows us to pass by reference

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

void CubeByReference(int&);

int main()
{
   int num = 5;

   cout << "Original value of num: " << num << endl;
   CubeByReference(num);
   cout << "New value of num: " << num << endl;

   return EXIT_SUCCESS;
}

void CubeByReference(int& n)
{
   n = n * n * n;
}
```


Memory Concepts

Pointer Basics

we can also effect passing by reference using pointers

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

void CubeBySimulatedReference(int *);

int main()
{
   int num = 5;

   cout << "Original value of num: " << num << endl;
   CubeBySimulatedReference(&num);
   cout << "New value of num: " << num << endl;

   return EXIT_SUCCESS;
}

void CubeBySimulatedReference(int *nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Memory Concepts

Pointer Basics

What will the following program display?

```
#include <iostream>
#include <cstdlib>
using namespace std;
void Goo(int *);
int main()
{
   int x = 0, *xPtr = &x;
   Goo(xPtr);
   cout << "Value of x is " << x << endl;
   return EXIT_SUCCESS;
}

void Goo(int *intPtr)
{
   *intPtr = 1;
}
```


More on Pointers

Pointer Arithmetic

- Pointers store memory addresses – must be properly handled
- Misuse of pointers can cause malfunctions not only to the errant program itself but also to other programs in the system
  - Pointers can be altered to point to anywhere in the system memory
  - If, for instance, a pointer is accidentally made to point to and operate on the memory used by other programs in the system, both the program and the entire system may crash
- Need to
  - *(in general)* master how to use pointers properly
  - *(specifically)* be careful in performing arithmetic operations on pointers to produce addresses that point to memory locations that are legal and intended

More on Pointers — Pointer Arithmetic

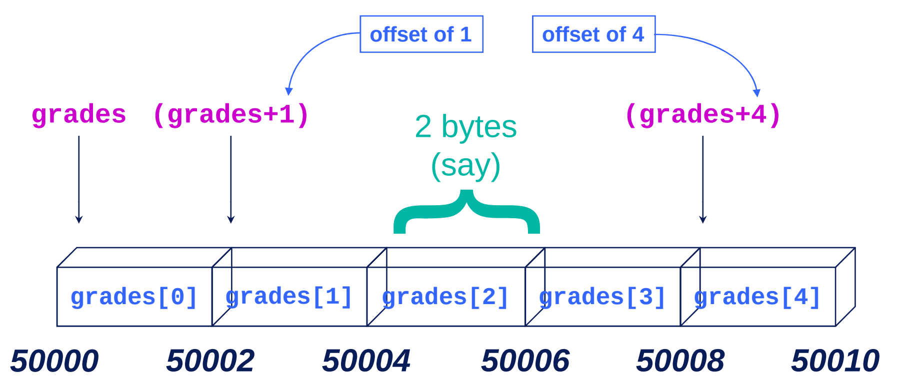
Integer values can be added to or subtracted from pointers (incrementing and decrementing included) to produce new addresses

- Value added to or subtracted from a pointer is automatically ***scaled*** so that the resulting address still points to a value of the correct type ➔ ***offset***
  - ***scale factor*** used is the ***number of bytes required to store the data type pointed to by the pointer*** ➔ one can use the `sizeof()` operator to obtain this number
- Example: for a pointer declared by `int x[5], *intPtr = x;`
  - `intPtr++;` or `++intPtr;`
    adds `sizeof(int)` to the address stored in `intPtr`
  - `intPtr--;` or `--intPtr;`
    subtracts `sizeof(int)` from the address stored in `intPtr`
  - `intPtr += intExp;` (`intExp` is some integer expression)
    adds `intExp * sizeof(int)` to the address stored in `intPtr`
  - `intPtr -= intExp;` (`intExp` is some integer expression)
    subtracts `intExp * sizeof(int)` from the address stored in `intPtr`

More on Pointers — Pointer Arithmetic

Recall that array names are really ***pointer constants***

```
int grades[5] = {98, 87, 92, 79, 85}; // 5 student grades
```



More on Pointers

Pointer Arithmetic

- The relational operators (**==**, **!=**, **<**, **>**, *etc.*) can be used to compare pointers that ***point to the same type***
  - Most common comparison operation is to use **==** and **!=** to determine if two pointers point to the same memory location
  - The ***null address*** (0, which can be assigned to a pointer to indicate that the pointer is not currently pointing to any memory location) is *type-independent* and may be compared with any pointer
- Operations that don't make sense (at least in most cases):
  - Add or subtract a *floating-point* constant to or from a pointer
  - Adding two or more pointers
  - Multiplication involving pointer(s)
  - Division involving pointer(s)
  - It is legal to perform subtraction on two pointers that point to the same data type – seldom useful, however

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98, 87, 92, 79, 85};
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << grades[i];
  return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98,87,92,79,85};
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << *(grades + i);
  return EXIT_SUCCESS;
}
```

array name is a pointer constant

MUST use parentheses in `*(grades + i)`

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98, 87, 92, 79, 85};
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << grades[i];
  return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98,87,92,79,85}, *gPtr = grades;
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << *(gPtr + i);
  return EXIT_SUCCESS;
}
```


More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98, 87, 92, 79, 85};
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << grades[i];
  return EXIT_SUCCESS;
}
```


not wrong but ...

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int i, grades[] = {98,87,92,79,85}, *gPtr = &grades[0];
  for (i = 0; i <= 4; ++i)
    cout << "\ngrades[" << i << "] = " << *(gPtr + i);
  return EXIT_SUCCESS;
}
```

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

`int grades[] = {98,87,92,79,85}, *gPtr = grades;`

gPtr

&grades[0]

gPtr is a ***pointer variable***
(address stored in it can be changed)

grades

&grades[0]

grades is a ***pointer constant***
(address stored in it cannot be changed)

grades[0]	grades[1]	grades[2]	grades[3]	grades[4]
or	*or*	*or*	*or*	*or*
*gPtr	*(gPtr+1)	*(gPtr+2)	*(gPtr+3)	*(gPtr+4)
or	*or*	*or*	*or*	*or*
*grades	*(grades+1)	*(grades+2)	*(grades+3)	*(grades+4)

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
   int nums[] = {16, 54, 7, 43, -5};
   int sum = 0, *numsPtr = nums;
   for (int i = 0; i <= 4; ++i)
      sum = sum + *numsPtr++;
   cout << "\nSum of all elements is "
        << sum << '.' << endl;
   return EXIT_SUCCESS;
}
```

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

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

int main()
{
   int nums[] = {16, 54, 7, 43, -5};
   int sum = 0, *numsPtr = nums;

   while (numsPtr <= nums + 4)
      sum += *numsPtr++;
   cout << "\nSum of all elements is "
        << sum << "." << endl;

   return EXIT_SUCCESS;
}
```


More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

program seen earlier that uses subscripted variable
(reproduced for comparison with one that uses pointers)

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

int FindMax(const int array[], int arraySize);

int main()
{
  int nums[5] = {2, 18, 1, 27, 16};
  int FindMax(int [], int);

  cout << "Maximum value of nums = " << FindMax(nums, 5);

  return EXIT_SUCCESS;
}

int FindMax(const int array[], int arraySize)
{
  int i, maxValue = array[0];

  for (i = 1; i < arraySize; ++i)
    if (maxValue < array[i])
      maxValue = array[i];

  return maxValue;
}
```

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

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

int FindMax(const int *arrayPtr, int arraySize);

int main()
{
  int nums[5] = {2, 18, 1, 27, 16};
  int FindMax(int *, int);

  cout << "Maximum value of nums = " << FindMax(nums, 5);

  return EXIT_SUCCESS;
}

int FindMax(const int *arrayPtr, int arraySize)
{
  int i, maxValue = *arrayPtr;

  for (i = 1; i < arraySize; ++i)
    if (maxValue < *(arrayPtr + i))
      maxValue = *(arrayPtr + i);

  return maxValue;
}
```

note that address operator (&) is not used because array name is already an address

More on Pointers

Pointer Arithmetic
Accessing Arrays Using Pointers

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

int FindMax(const int *arrayPtr, int arraySize);

int main()
{
  int nums[5] = {2, 18, 1, 27, 16};
  int FindMax(int *, int);

  cout << "Maximum value of nums = " << FindMax(nums, 5);

  return EXIT_SUCCESS;
}

int FindMax(const int *arrayPtr, int arraySize)
{
  int i, maxValue = *arrayPtr++;

  for (i = 1; i < arraySize; ++i, ++arrayPtr)
    if (maxValue < *arrayPtr)
      maxValue = *arrayPtr;

  return maxValue;
}
```

style giving identical results but not as clear or readable (thus to be avoided)

Memory Management — Dynamic Memory Allocation

- Fixed-size arrays as defined in statements like

  ```
  double array[20];
  ```

  have two drawbacks:
  - memory is wasted if array size exceeds the number of values to be stored
  - must recompile if array size is less than the number of values to be stored
- The above drawbacks stem from...
  - memory being allocated at ***compile time*** (when the program is compiled)
- To overcome the above drawbacks, a mechanism to allocate memory at run-time is needed ➔ dynamic (***run-time***) memory allocation
- Two basic operations for dynamic memory allocation in C++:
  - allocate/acquire/request *new* dynamic memory when it is needed ➔ `new`
  - deallocate/release/free *previously allocated* dynamic memory when it is no longer needed ➔ `delete`

Memory Management — Dynamic Memory Allocation

To dynamically request a block of memory large enough to hold an ***object*** of type `int` (say)...

```
int *intPtr = 0;
...
intPtr = new int;
...
```

- if request can be granted, `new` returns address of memory allocated
- (for pre-standard compilers) if request cannot be granted, `new` returns **0** (the ***null address***) ➔ failure in memory allocation can then be trapped as in

```
if (intPtr == 0)
{
   cerr << "\n*** No more memory ***\n";
   exit(EXIT_FAILURE);
}
```

  - (standard C++ also allows such use) if `new` is replaced with `new(nothrow)`
- (standard C++) uses "exception handling" to deal with allocation failure

Memory Management — Dynamic Memory Allocation

- To free memory blocks previously acquired (but no longer needed)...

```
int *intPtr = 0;
...
intPtr = new int;
...
delete intPtr;
intPtr = 0;
...
```

  - Note that **delete** only frees up memory pointed to by **intPtr** ➔ it does NOT in any way delete **intPtr** itself
- Can also initialize object while it is dynamically created, for *e.g.*:

```
float *piPtr = new float(3.14159);
```

- In practice, **new** is rarely used to allocate space for individual instances of basic types (**int**, **char**, **float**, **double**, *etc.*)
  - mainly used to allocate space for user-defined data types (which can be huge)

Memory Management — Dynamic Memory Allocation

- To dynamically (*i.e.*, at run-time) allocate memory for an ***array*** of **int** (say) that will hold up to 5 (say) elements...

```
int *intArrayPtr = new int[5];
```

(or using separate statements)

```
int *intArrayPtr = 0;
intArrayPtr = new int[5];
```

- To free up (release) the above dynamically allocated memory...

```
delete [] intArrayPtr;
```

- Example code segment:

```
cout << "Enter number of entries to process: ";
cin >> numEntries;
double *dblArrayPtr = new(nothrow) double[numEntries];
if (dblArrayPtr == 0)
{
   cerr << "Error allocating memory..." << endl;
   exit(EXIT_FAILURE);
}
...
delete [] intArrayPtr;
...
```

Dynamic Memory Allocation

Example Application

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

int main()
{
   char *charArrayPtrHold = 0, *charArrayPtr = 0, oneChar;
   int count = 0;

   cout << "Enter some text below:" << endl;
   cin.get(oneChar);
   while (oneChar != '\n')
   {
      count++;
      charArrayPtrHold = charArrayPtr;
      charArrayPtr = new(nothrow) char [count];
      if (charArrayPtr == 0)
      {
         cerr << "Error allocating memory..." << endl;
         exit(EXIT_FAILURE);
      }
```

(continued)

Dynamic Memory Allocation

Example Application

```
      for (int i = 0; i < count - 1; i++)
         *(charArrayPtr + i) = *(charArrayPtrHold + i);
      *(charArrayPtr + count - 1) = oneChar;
      delete [] charArrayPtrHold;
      charArrayPtrHold = 0;
      cin.get(oneChar);
   }

   cout << "The text you entered in reverse is:" << endl;
   for (int j = count - 1; j >= 0; j--)
      cout << *(charArrayPtr + j);
   cout << endl;

   delete [] charArrayPtr;
   charArrayPtr = 0;

   return EXIT_SUCCESS;
}
```

Though not wrong, the code in this example is inefficient.
Can you see why?
How'd you make it more efficient?

Dynamic Memory Allocation — Pitfall: Dangling Pointer

What's Wrong with the Following Code Segment?

```
int *ptr1 = new int;
if (ptr1 == 0) // for pre-standard compilers
{
   cerr << "Error allocating memory..." << endl;
   exit(EXIT_FAILURE);
}
int *ptr2 = 0;
*ptr1 = 100;
ptr2 = ptr1;
cout << "*ptr2 = " << *ptr2 << endl;
delete ptr2;
ptr2 = 0;
cout << "*ptr1 = " << *ptr1 << endl;
```


Dynamic Memory Allocation — Pitfall: Memory Leak

What's Wrong with the Following Code Segment?

Assume all variables have been properly declared

```
do
{
   intPtr = new int [10];
   if (intPtr == 0) // for pre-standard compilers
   {
      cerr << "Error allocating memory..." << endl;
      exit(EXIT_FAILURE);
   }
   // ... dynamic array is used here to solve a problem

   cout << "Do another (y or n)? ";
   cin >> answer;
}
while (answer != 'n' && answer != 'N');
```

Textbook Readings

- Chapter 2
  - Page 65 (Default Arguments)
- Chapter 4
  - Section 4.1 (Pointers and Dynamic Memory)
  - Section 4.2 (Pointers and Arrays as Parameters)