CSCI 3132 – Object Orientation and Generic Programming

Week 4 – C++ Programming Basics

C++ Language Reference: https://en.cppreference.com

Simple Data Types in C++

C++ Data Types

- C++ Fundamental Data Types
  - Character
    - char
    - char16_t
    - char32_t
    - wchar_t
  - Integral (signed & unsigned)
    - short
    - int
    - long
    - long long
  - Floating-point
    - float
    - double
    - long double
  - Others
    - bool
    - void
    - nullptr

C++ Data Types

- 1 byte = 8 bits (normally),
- Size of data types represented in bytes
  - Sometimes, size mentioned as “at least x”
    - Example: int -> at least 16-bits
- C++ reference guarantees that:

  1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- For 64-bit bytes, all types (including char) are 64-bits wide
  - sizeof(…) returns 1 for every type

Example

```
#include <iostream>
int main()
{
    char ch;
    short s;
    int i;
    long l;
    float f;
    double d;
    std::cout << "char = " << sizeof(ch)
        << "\nshort = " << sizeof(s)
        << "\nint = " << sizeof(i)
        << "\nfloat = " << sizeof(f)
        << "\nlong = " << sizeof(l)
        << "\ndouble = " << sizeof(d)<<"\n";
    system("pause");
}
```

Output from cpp.sh

```
char = 1
short = 2
int = 4
float = 4
long = 8
double = 8
```

Output from Visual C++

```
char = 1
short = 2
int = 4
float = 4
long = 4
double = 8
```

Example

1. What should be the data type of fact?
2. What is the output of this code?

```
void main() {
________ fact = 1;
for (int i = 1; i <= 20; i++)
{
      fact *= i;
}
cout << "\nFactorial = " << fact << " and
size of variable = "<<sizeof(fact)<<"\n";
```

Solution

Following is the output for int, long and long long

int

```
Factorial = -2102132736 and size of variable = 4
```

long

```
Factorial = -2102132736 and size of variable = 4
```

long long

```
Factorial = 2432902008176640000 and size of variable = 8
```

Variable Declarations

- Following are all valid

```
int a, b, c;
```

```
int a;
int b;
int c=5;
```

```
int a = 0, b, c=5;
```

Variable Initialization

- Variables can be initialized when they are declared
  - In C++, there are three ways to initialize a variable

```
int x = 0;
```

```
int x(0);
```

```
int x{0};
```

```
int a = 5;            // initial value: 5
int b(3);             // initial value: 3
int c{ 2 };           // initial value: 2
int result;           // initial value undetermined
```

Type Deduction

- "auto" can be used as the type specifier for a variable

```
float foo = 0.0;
auto bar = foo; //same as: float bar = foo;
```

- Uninitialized variables can also make use of type deduction using "decltype"

```
float foo = 0.0;
decltype(foo) = bar; //same as: float bar;
```

- Used either when:
  - type cannot be obtained by other means, or
  - when using it improves code readability

typedef Declarations

- You can create a new name for an existing type using `typedef`
  - Example :

```
typedef int inch;
inch distance;
```

- Since C++ 11, “typedef” replaced with “**using**”

```
using inch = int;
```

extern

`**extern**' keyword means that a symbol can be accessed, but not defined. It should be defined (as a global) in some other module.

```
// Variable declaration:
extern int a, b;
int main ()
{
  // Variable definition:
  int a, b;
  int c;
  // actual initialization
  a = 10;
  b = 20;
  c = a + b;
  cout << c << endl ;
}
```

Function Declaration

- Function name can similarly be provided at the time of its declaration and its actual definition can be given elsewhere.

```
// function declaration
int my_func();

int main()
{
    // function call
    int i = my_func();
}
// function definition
int my_func()
{
    return 0;
}
```

l-values and r-values

- There are 2 kinds of expressions in C++
  - **l-value**
    - Expressions that refer to a memory location
    - An l-value may appear at either the left-hand or right-hand side of an assignment
    - E.g. int i = 20; x=y;
  - **r-value :**
    - A data value that is stored at some address in memory
    - An expression that cannot have a value assigned to it
    - An r-value may appear on the right- but not left-hand side of an assignment.
    - E.g. 20=x would be incorrect, so would 20=20;

C++ Constants and Literals

- Fixed values that the program may not alter
- Can be of any of the basic data types
  - Integer, numeric, float, char, string, Boolean
  - Integer literals can be decimal, octal or hexadecimal
    - Prefix of 0x for hexadecimal and 0 for octal is used
    - E.g. int x=077 assigns the value 63 to x
int x=0xA0 assigns the value 160 to x
int x=079 invalid as octal numbers are 0-7
  - Floating point can be decimal or exponential
    - E.g. 3.14159, 314159E-5L are valid
314159E, .e55 are invalid

C++ Constants and Literals

- Boolean literals
  - true and false
  - Should not consider value of 1 equal to true
- Character literals
  - Can be
    - a plain character (e.g. ‘x’),
    - an escape sequence (e.g. ‘\n’),

C++ Constants and Literals

- Defining constants
  - Use #define preprocessor
    - E.g.
      ```
      #define DICE_SIDES 6
      #define NEWLINE '\n'
      #define FALSE true    //legal but terrible
      ```
    - #define is a pre-processor directive
  - Use const keyword
    - E.g.
      ```
      const int DICE_SIDES = 6;
      const char NEWLINE = '\n';
      ```
    - const declares an actual variable

Compound Data Types in C++

Compound Data Types

- Arrays
- Character sequences
- Strings
- Pointers
- Dynamic memory
- Data structures
- Other data types

Arrays

- A series of elements of the same type placed in contiguous locations in memory
  - Can be individually referenced by their index
  - Are blocks of static memory whose size must be determined at compile time
- Typically declared as:

```
type name [elements]
```

- Example – Array of 6 integers:

```
int nums [6]
```

  - Need a constant expression to represent the number of array elements

Array Initialization

- Arrays can be explicitly initialized with no values or to specific values

```
int nums[6] = {}; //initialized to 0
int nums[6] = {12, 2, 30, 1, 4, 7};
int nums[] = {1, 2, 3}; //array size 3
int nums[] {10, 20, 30};
```

- Array elements can be accessed using their index
  - Arrays index starts from 0
  - Example:

```
nums[0] = 20; //set 1st element to zero
cout << nums[1] + nums[3];
//results in output of 2+1 = 3
```

Array Example

- What does the following code segment do?

```
int foo [] = {1, 2, 3, 4, 5, 6};
int n, result=1;

int main ()
{
  for ( n=0 ; n<6 ; ++n )
  {
    result *= foo[n];
  }
  cout << result;
  return 0;
}
```

Multidimensional Arrays

- Arrays of arrays

```
int foo [2][3];
//creates a 2x3 array

cout<<foo[1][0]
//outputs 1st element of 2nd row
```

- Study the following multi-dimensional array

```
char test[100][365][24][60][60]
```

  - What does it do?
  - How much memory would it consume?

Arrays as Parameters

- Arrays passed by address as function arguments
  - Much faster and more efficient than passing a block of memory
  - Parameter declared as array with empty brackets

```
void func (int arr[]);
…
int l_arr[10];
…
func(l_arr);
```

- For multidimensional arrays, necessary to specify depth of dimensions

```
void func (int arr[][2][3]);
```

Arrays

- Since C++ 11, std::array is a container that encapsulates fixed size arrays.

```
#include <iostream>
#include <array>

int main()
{
std::array<int,3> myarray {10,20,30};
for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];
for (int elem : myarray)
    std::cout << elem << '\n';

}
```

https://en.cppreference.com/w/cpp/container/array

Character Sequences

- Expressed as an array of characters

```
char foo [20];
```

- End of sequence represented by *null character* '\0'

```
char foo[] = {'H', 'e', 'l', 'l', 'o', '\0'};
//can also be initialized as:
char foo[] = "Hello";
```

  – Size of array = 6 elements (including null character)

Character Sequences

- Consider the following declaration:

```
char foo[] = "Hello";
```

  – Which of the following assignments would be valid?

```
foo = "Hi!";
```

```
foo[] = "Hi!";
```

```
foo = {'H', 'i', '!', '\0'};
```

```
foo[0] = 'H';
foo[1] = 'i';
foo[2] = '!';
foo[3] = '\0';
```

Exercise

- Write a program to copy the character sequence from my_char_arr to your_char_arr

```
char my_char_arr[] = "Object Orientation";
```

- Solution

```
int main()
{
   char my_char_arr[]  = "Object Orientation";
   char your_char_arr[19];
   for (int i = 0; i < 19; ++i) {
         your_char_arr [i] = my_char_arr[i];
   }
   cout << your_char_arr;
   return 0;
}
```

Useful Functions – C-style Character String

Function	Purpose
`strcpy(s1, s2);`	Copies string s2 into string s1.
`strcat(s1, s2);`	Concatenates string s2 onto the end of string s1.
`strlen(s1);`	Returns the length of string s1.
`strcmp(s1, s2);`	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
`strchr(s1, ch);`	Returns a pointer to the first occurrence of character ch in string s1.
`strstr(s1, s2);`	Returns a pointer to the first occurrence of string s2 in string s1.

C++ `string` Class

- Strings are objects that represent sequences of characters
  - Part of the `std` namespace
- Initializing

```
string my_str (“Hello World”);
```

- I/O

```
cin >> my_str;        //reads a string
```

```
getline(cin, my_string, '\n');
//reads a line terminated by \n
```

C++ `string` Class

- String concatenation

```
string my_str1 = "Hello ";
string my_str2 = "World";
string my_str3 = my_str1 + my_str2;
cout<<my_str3<<endl; //output "Hello World"
```

- String comparison

```
string my_str1 = "Hello ";
string my_str2 = "World";
if (my_str1 != my_str2)
   cout<<my_str1 + my_str2<<endl;
//output "Hello World"
```

- What would happen if != is replaced by < or >

Some useful `string` functions

`size`	Return length of string
`length`	Return length of string
`max_size`	Return maximum size of string
`empty`	Test if string is empty
`Operator []`	Get character of string
`at`	Get character in string
`back`	Access last character
`front`	Access first character
`copy`	Copy sequence of characters from string
`find`	Find content in string
`find_first_of`	Find character in string
`find_last_of`	Find character in string from the end
`substr`	Generate substring
`compare`	Compare strings

Pointers

Memory Locations

Q: What happens when you define a variable, e.g. int x?

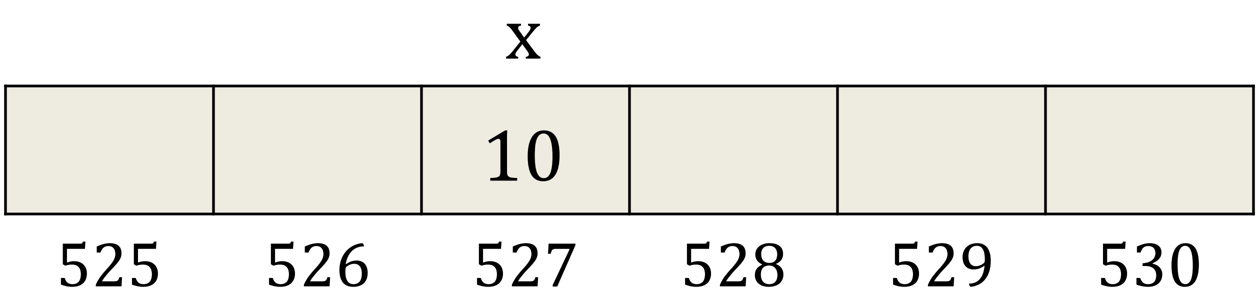
A: It is assigned a space in memory as shown below

Q: What value does it hold?

A: Any value present at that memory location (non-static)

Q: What happens when you initialize it (e.g. x=10)?

A: The value is written in the memory allocated to it



Address of a Variable

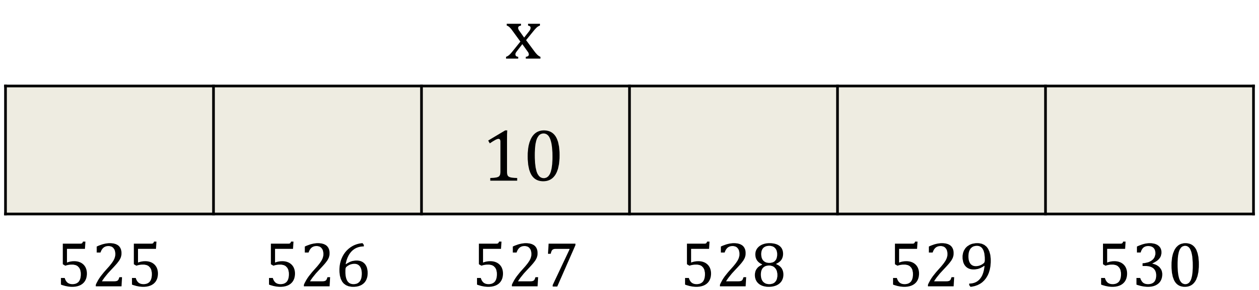
Q: How do you get the memory address of that variable?

A: Using the Address-of operator - **&**

Example: &x returns 527

Q: Where can we store this address of a variable?

A: In a **pointer**!



Pointers

Q: What is a pointer?

A: variable that stores memory addresses, usually of other variables

Q: Why do we need pointers?

A: - dynamically allocate memory – you can write programs that can handle unlimited amounts of memory

- allow a function to modify a variable passed to it

- easier to pass around the location of a huge amount of data than passing the data itself

Pointers

Q: How do you define a pointer?

A: <*variable type*> * *ptr_name*;

Example: int * y;

- Note that the pointer's type is not *int*, but rather the variable that the pointer points to is *int*
- pointer's definition needs to include the data type it is going to point to
  - Example:

```
int x;
int * y;
y = &x;
```

Here, y is the pointer variable and contains the memory address of the integer variable x

Dereference Operator – *

Q: How can we get the content of the memory address pointed to by the pointer?

A: By using the dereference operator '*'

Note: Read the * operator as "value pointed to by"

Example:

```
int x = 10;        //variable x = 10
int * y = &x;      //pointer variable y = Address of x
int z = *y;        //z = "value pointed to by y" = x, i.e. z = x;
```

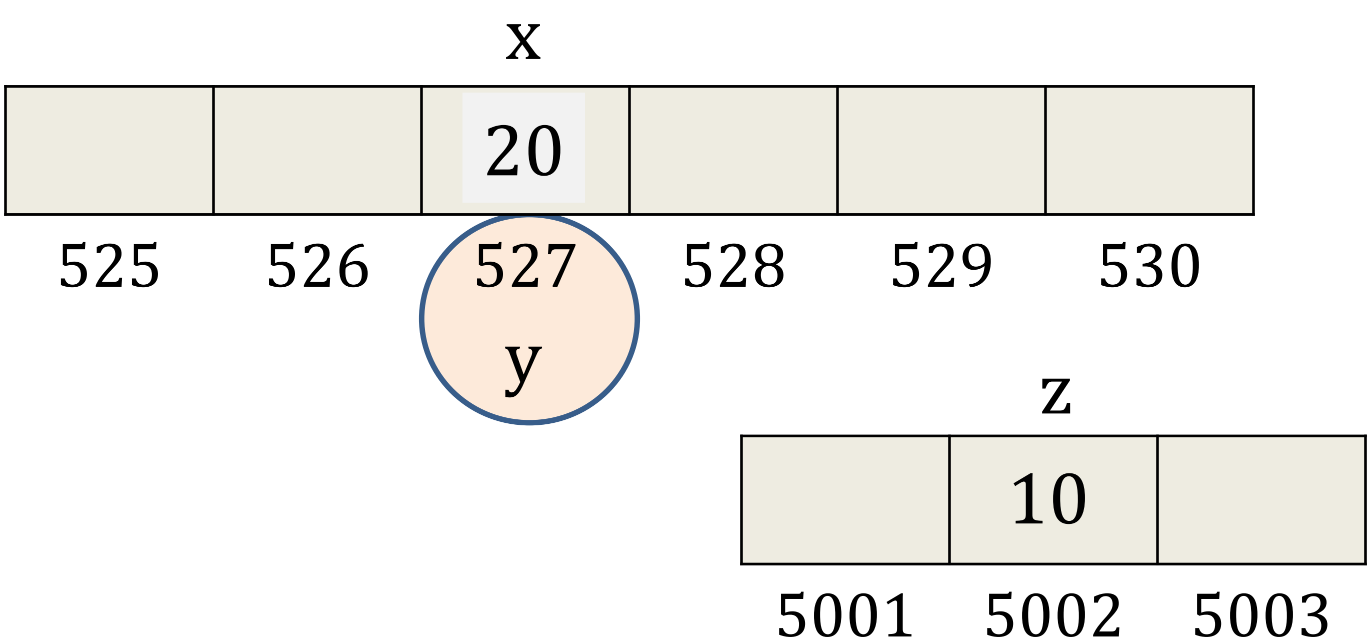
Pointer versus Dereference Operator

int x = 10; //variable x = 10

int * y = &x; //pointer variable y = Address of x

int z = *y; //z = “value pointed to by y” = x, i.e. z = x;

*y = 20; //”value pointed to by y” = 20;



Example

```
#include <iostream>
int main() {
    int x1, x2;
    int * p1;
    p1 = &x1;     //p1 = address of x1
    *p1 = 10;     //value pointed to by p1=> x1=10
    p1 = &x2;     //p1 = address of x2
    *p1 = 20;     //value pointed to by p1=> x2=20
    std::cout<<"\nValue of x1 = "<<x1;
    std::cout<<"\nValue of x2 = "<<x2;
    return 0;
}
```

```
Value of x1 = 10
Value of x2 = 20
```

Exercise

```
#include <iostream>
int main() {
    int x1=10, x2=20;
    int * p1, * p2;
    p1 = &x1;
    p2 = &x2;
    *p1 = 100;
    *p2 = *p1;
    p1 = p2;
    *p1 = 200;
    std::cout<<"\nValue of x1 = "<<x1;
    std::cout<<"\nValue of x2 = "<<x2;
    return 0;
}
```

Exercise – Solution Explained

```
#include <iostream>
int main() {
    int x1=10, x2=20;
    int * p1, * p2;
    p1 = &x1;           //p1 = address of x1
    p2 = &x2;           //p2 = address of x2
    *p1 = 100;          //value pointed to by p1=100 => x1=100
    *p2 = *p1;          //value pointed to by p2=value pointed to by p1=> x2=100
    p1 = p2;            //p1 points to same addr as p2 i.e. x2
    *p1 = 200;          //value pointed to by p1=200 => x2=200
    std::cout<<"\nValue of x1 = "<<x1;
    std::cout<<"\nValue of x2 = "<<x2;
    return 0;
}
```

```
Value of x1 = 100
Value of x2 = 200
```

Pointers and Arrays

- Array can be implicitly converted to the pointer of the proper type
  - Example:
    ```
    int arr[20];
    int * ptr;
    ptr = arr;
    ```
  - Note however that `arr=ptr` would be invalid
- An array can be used just like a pointer to its first element
  - Pointers and arrays support the same set of operations
  - Pointers, however, can be assigned new addresses, while arrays cannot

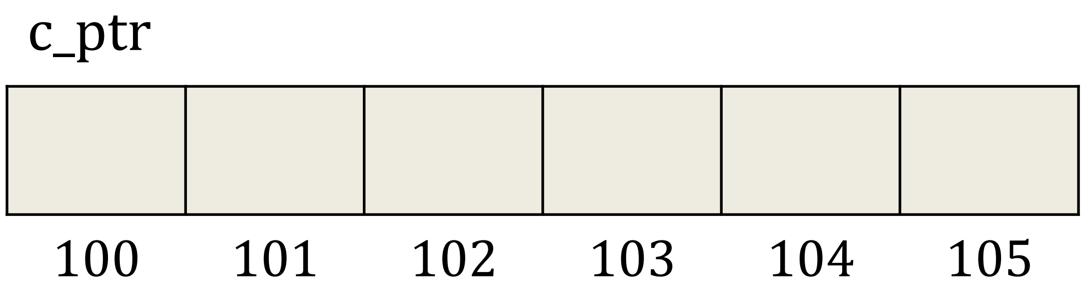
Example

```cpp
#include <iostream>
int main() {
    char arr[5];
    char * ptr;
    ptr = arr;  *ptr = 'H';
    ptr++;  *ptr = 'E';
    ptr = &arr[2];  *ptr = 'L';
    ptr = arr + 3;  *ptr = 'L';
    ptr = arr;  *(ptr + 4) = 'O';
    for (int n = 0; n<5; n++)
        std::cout << arr[n];
}
```

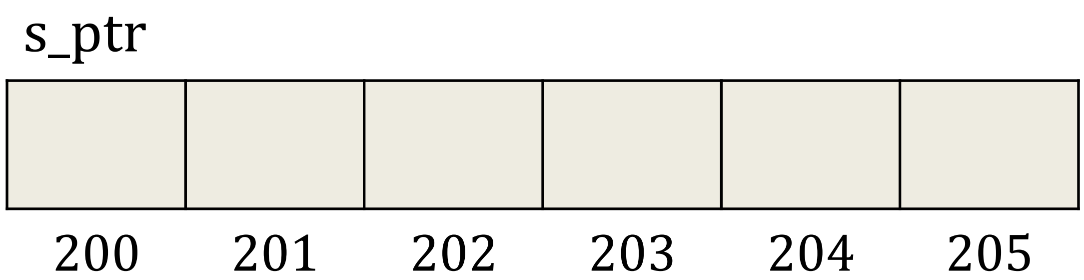
```
HELLO
```

Pointer Arithmetic

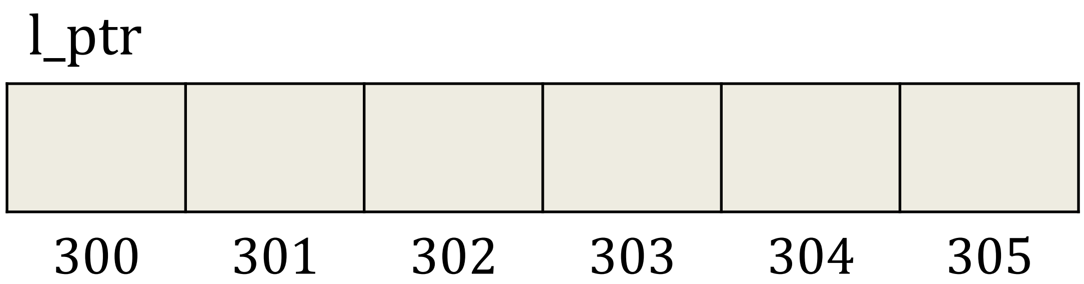
```
char * c_ptr;
++c_ptr;
```




```
short * s_ptr;
++s_ptr;
```




```
long * l_ptr;
++l_ptr;
```



Pointer Arithmetic

- Only addition and subtraction operations allowed on pointers
  - What happens in the following cases?

```
*p++     → *(p++)  //value pointed to by p, then increment pointer
*++p     → *(++p)  //increment pointer, then value pointed to by p
++*p     → ++(*p)  //increment value pointed to by p
(*p)++   → (*p)++  //value pointed to by p, then increment value
```

```
*p++ = *q++     → *p = *q; ++p; ++q
```

Pointer Arithmetic – Example

```
#include <iostream>
using namespace std;
int main() {
    char ch[] = "Hello";
    char * p = ch;
    char * q = ch;
    cout <<*p<<"\t"<<*p++<<"\n"<<*q<<"\t"<<*++q<<"\n";
    (*q++);
    cout <<*p<<"\t"<<++*p<<"\n"<<*q<<"\n";
}
```

```
H   H
H   e
e   f
l
```

Pointer as Function Argument

```cpp
#include <iostream>
void chg_ptr(int* ptr)
{
    *ptr = 200; //change value of ptr
}
int main()
{
    int n = 1;
    int *p = &n;
    std::cout << *p << "\n";    //outputs 1
    chg_ptr(p);      //change value pointed to by p
    std::cout << *p << "\n";    //outputs 200
    return 0;
}
```

```
1
200
```

const Pointers

- Pointers can themselves be declared const
  - Cannot change the address assignment

```
int n = 1, x=2;
int * const p = &n;
*p = 500; //valid
p = &x;   //invalid
p++;      //invalid
```

const Pointers as Arguments

```
#include <iostream>
using namespace std;
void chg_ptr(int* ptr)
{
    *ptr = 200; //change value of ptr
}
void const_ptr(const int* ptr)
{
    *ptr = 200; //error
}
int main()
{
    int n = 1;
    int *p = &n;
    cout << *p << endl; //outputs 1
    chg_ptr(p);         //change value pointed to by p
    cout << *p << endl; //outputs 200
    const_ptr(p);       //Error!
    return 0;
}
```

Dynamic Memory

- Earlier we saw that regular arrays need to be fixed size so that memory could be allocated at compile time
  - Array size cannot be variable or dynamic
  - Array size cannot be based on user input
- C++ provides a way to dynamically allocate memory
  - Dynamic memory allocation done using **`new`** and **`delete`**
  - Memory is allocated on heap rather than the stack
  - Memory may not be allocated if not enough memory available
  - Example: int * ptr = new int; //allocates memory address of size int
    - This memory should be freed at the end of its usage

Dynamic Memory

```
#include <iostream>
using namespace std;
int main() {
    int i, n;
    cout << "Enter size of array: ";
    cin >> i;
    int * p = new (nothrow) int[i]; //may need header <new>
    if (p == nullptr)
        cout << "Error: memory could not be allocated";
    else
    {
        for (n = 0; n < i; n++)
        {
            p[n] = n;
            cout << p[n];
        }
    }
    delete[] p;
    return 0;
}
```

Dynamic Memory Allocation for Objects

```cpp
#include <iostream>
class Box
{
   public:
      Box() {
         std::cout << "Constructor called!\n";
      }
      ~Box() {
         std::cout << "Destructor called!\n";
      }
};
int main( )
{
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array
   return 0;
}
```

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Exercise

- Write a function that takes the array size and a dynamically allocated pointer to that array as argument, and prints the contents of the array

Exercise – Sample Solution

```
#include <iostream>
using namespace std;
void Func(int n, int* p){
    for (int i=0; i<n; i++,p++){
        cout<<*p<<" ";
    }}
int main(){
    int i;
    cout << "Enter size of array: ";
    cin >> i;
    int * p = new (nothrow) int[i]; //may need header <new>
    for (int n = 0; n < i; n++) {  p[n] = n; }
    Func(i,p);
    delete[] p;
    return 0;
}
```

Fun with Pointers – 1

```
#include <iostream>
using namespace std;
int main(){
    int i = 3;
    int *j;
    int **k;
    j=&i;
    k=&j;
    cout<<k<<" "<<*k<<" "<<**k<<endl;
    return 0;
}
```

Fun with Pointers – 1

```
#include <iostream>
using namespace std;
int main(){
    int i = 3;
    int *j;
    int **k;
    j=&i;
    k=&j;
    cout<<k<<" "<<*k<<" "<<**k<<endl;
    return 0;
}
```

```
0x7fffe24d1a58 0x7fffe24d1a54 3
```

Fun with Pointers – 2

```
#include <iostream>

int main(){
    int i = 5;
    int *p;
    p = &i;
    std::cout<<*&p<<" "<<&*p<<"\n";
    return 0;
}
```

Fun with Pointers – 2

```
#include <iostream>

int main(){
    int i = 5;
    int *p;
    p = &i;
    std::cout<<*&p<<" "<<&*p<<"\n";
    return 0;
}
```

```
0x7ffc53f55edc 0x7ffc53f55edc
```

Fun with Pointers – 3

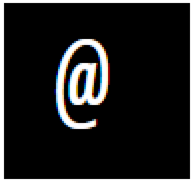
```
#include <iostream>

int main(){
    short a = 320;
    char *ptr;
    ptr =( char *)&a;
    std::cout<<*ptr<<"\n";
    return 0;
}
```

Fun with Pointers – 3

```cpp
#include <iostream>

int main(){
    short a = 320;
    char *ptr;
    ptr =( char *)&a;
    std::cout<<*ptr<<"\n";
    return 0;
}
```



Identify What's Wrong

```
string *getName() {
        string fullName[3];
        cout << "Enter first name: ";
        getline(cin, fullName[0]);
        cout << "Enter middle initial: ";
        getline(cin, fullName[1]);
        cout << "Enter last name: ";
        getline(cin, fullName[2]);
        return fullName;
}
```

Function returns a pointer to an array that no longer exists

Returning Pointers from a Function

Return a pointer from a function only if it is a pointer to:

- An item that was passed in as a function argument
  - E.g. `string *getName(string fullName[])`
- A dynamically allocated chunk of memory
  - E.g. `string *fullName = new string[3];`

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2)  // compares
                     // contents
```

Using Smart Pointers to Avoid Memory Leaks

- In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it.
- Three types of smart pointer:

```
unique_ptr
shared_ptr
weak_ptr
```

- Must `#include` the memory header file:

```
#include <memory>
unique_ptr<int> ptr( new int );
```

  - The notation `<int>` indicates that the pointer can point to an `int`.
  - The name of the pointer is `ptr`.
  - The expression `new int` allocates a chunk of memory to hold an `int`.
  - The address of the chunk of memory will be assigned to `ptr`.