Linked Data Representation in C++

Dr. Robert Amelard

ramelard@uwaterloo.ca

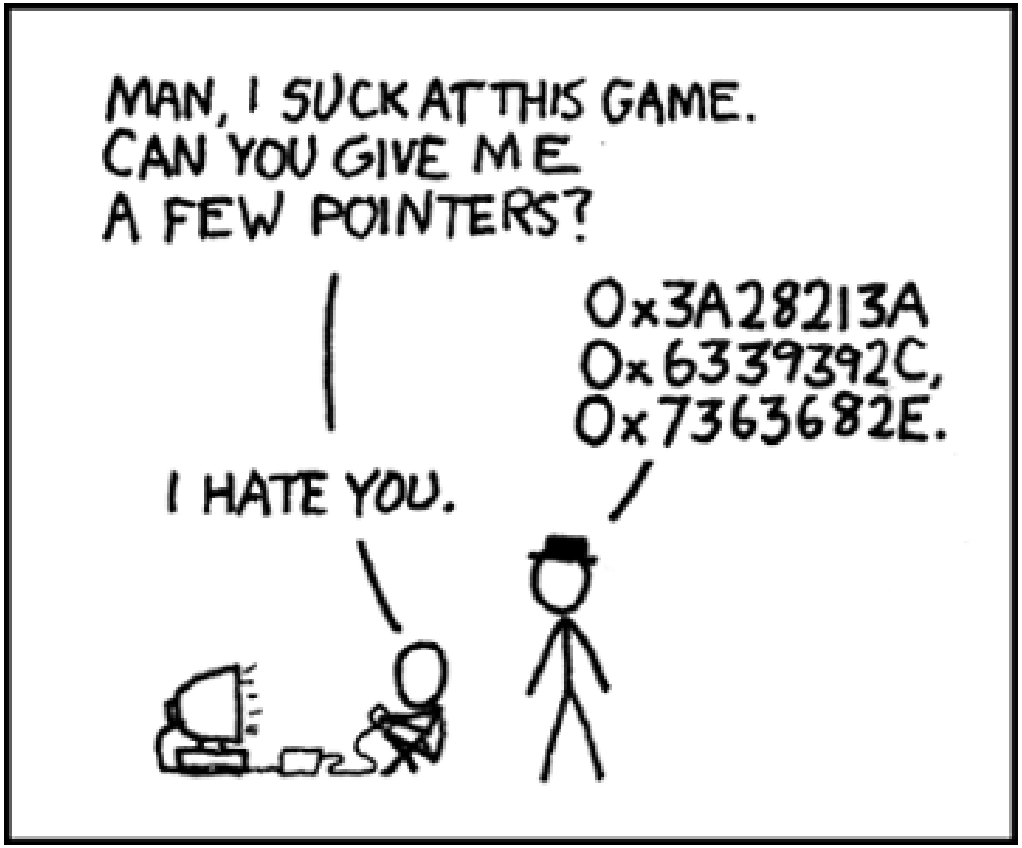
Objectives

Core Content:

- **A Program “Under the Hood”**
- **C++ Memory Model**
- **Classes and Dynamic Memory**
- **Dynamic Arrays**
- **Separate Compilation Units**

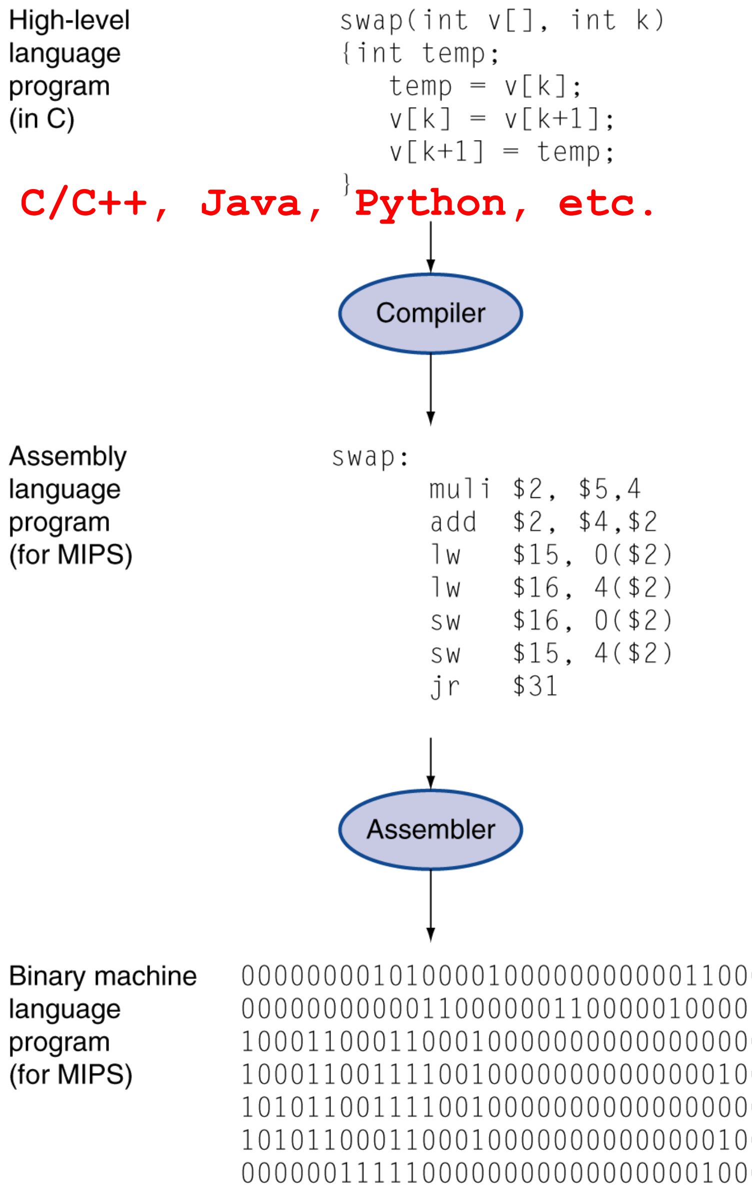
Additional Information:

- Namespaces via `using` Directives
- Global and Unnamed Namespaces
- Nested Namespaces


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Levels of Program Code

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware language**
  - Binary digits (bits)
  - Encoded instructions and data



Program /1

- A **program** boils down to a sequence of instructions that can be interpreted and executed by a computer
- **Fundamentally, a computer stores a program in memory, and the processor (CPU) runs the instructions one line at a time.**

```
instruction1
instruction2
instruction3
instruction4
...
```

Firefox.exe

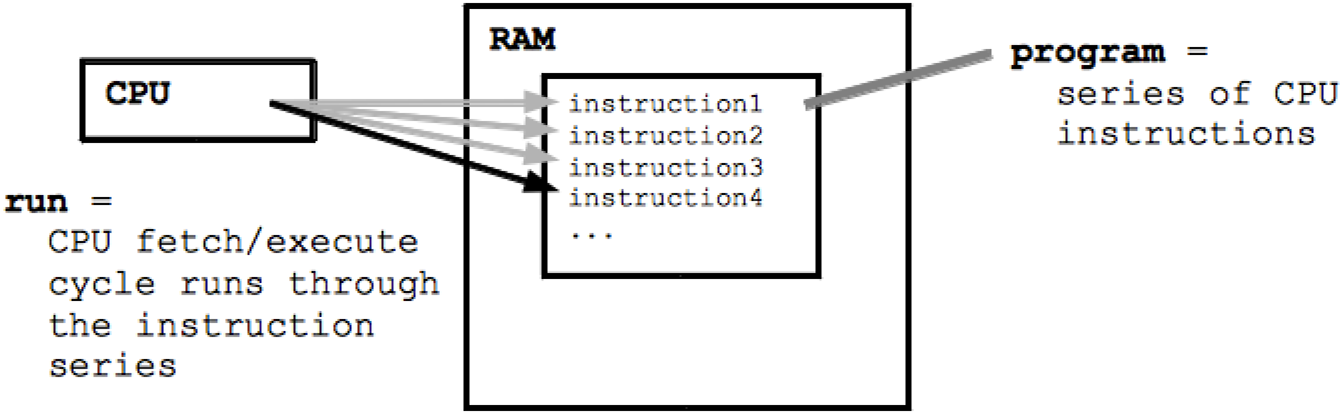
ECE 350 LAB,LEC,TUT 0.50 Course ID: 015297

Real-Time Operating Systems

Memory/virtual memory and caching; I/O devices, drivers, and permanent storage management; process scheduling; queue management in the kernel; real-time kernel development. Aspects of multi-core operating systems. [Offered: F, W, first offered Fall 2020]

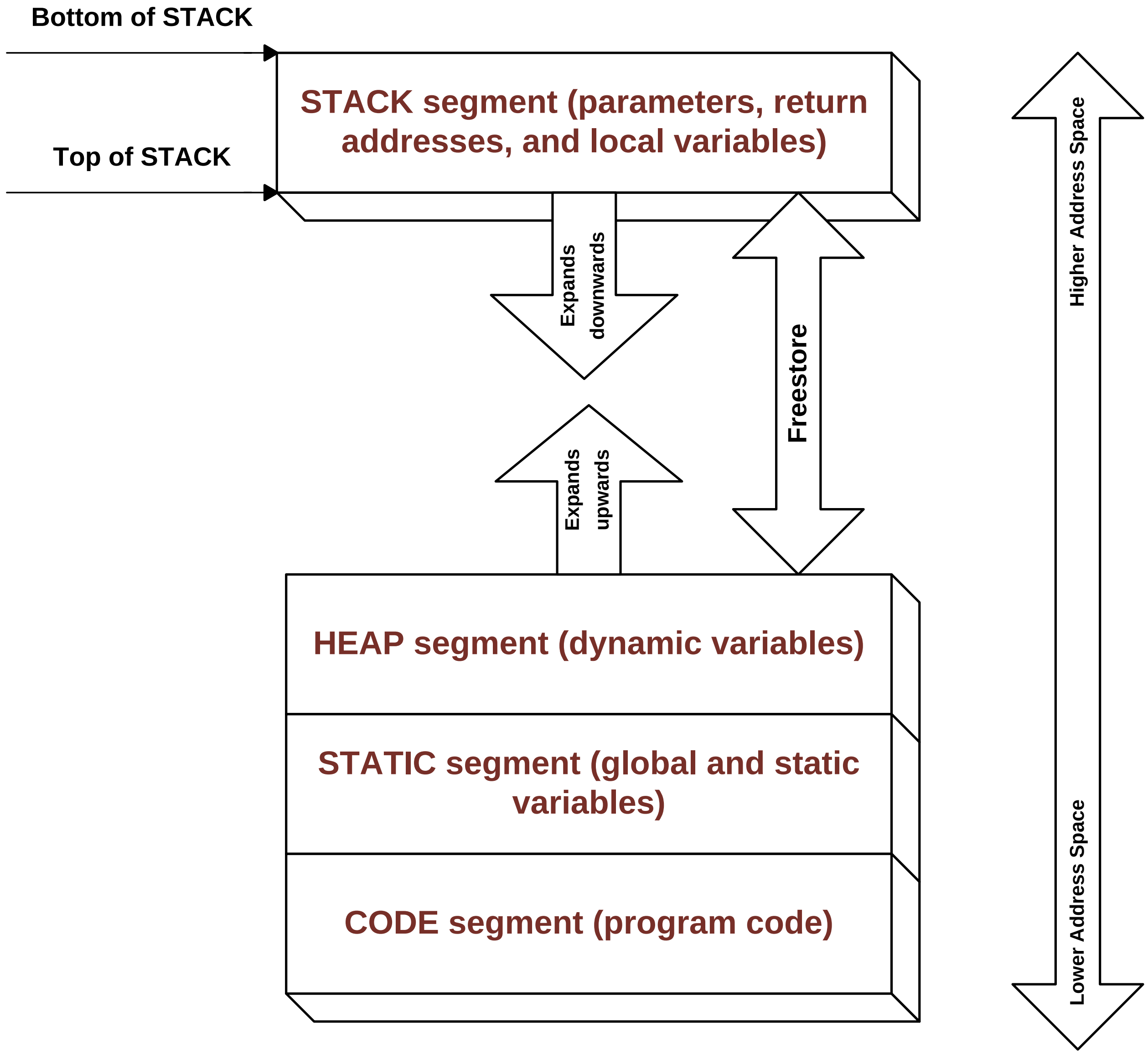
Program /2

- What happens when you “run” a program
  - (Before: the program is **compiled** into machine code)
  - The operating system allocates a space in memory (RAM)
  - The program instructions are copied from storage (disk drive) to RAM.
  - The CPU is pointed at the first instruction (**main()**)



https://web.stanford.edu/class/cs101/software-1.html

Program Memory Map



Memory Management

- **Memory Heap**
  - Reserved for dynamically-allocated variables
  - All new dynamic variables consume memory in freestore
  - If too many, they could use up all the freestore memory
  - Future "`new`" operations will fail if freestore is full

- **Freestore size is typically large**
  - Most programs will not use up all the memory
  - Memory management is still important
  - Crucial software engineering principle
  - Memory IS finite, regardless of how much of it is available

Memory leak

Dynamic and Static/Local Variables

- **Dynamic variables**
  - Created with `new` operator
  - Created and destroyed while program runs
- **Static/local variables**
  - Declared within scope (e.g., function definition, class instance)
  - Not dynamic
  - Created at beginning of scope (e.g., function is called)
  - Destroyed when program goes out of scope (e.g., function call completes)

Static vs Dynamic – Example

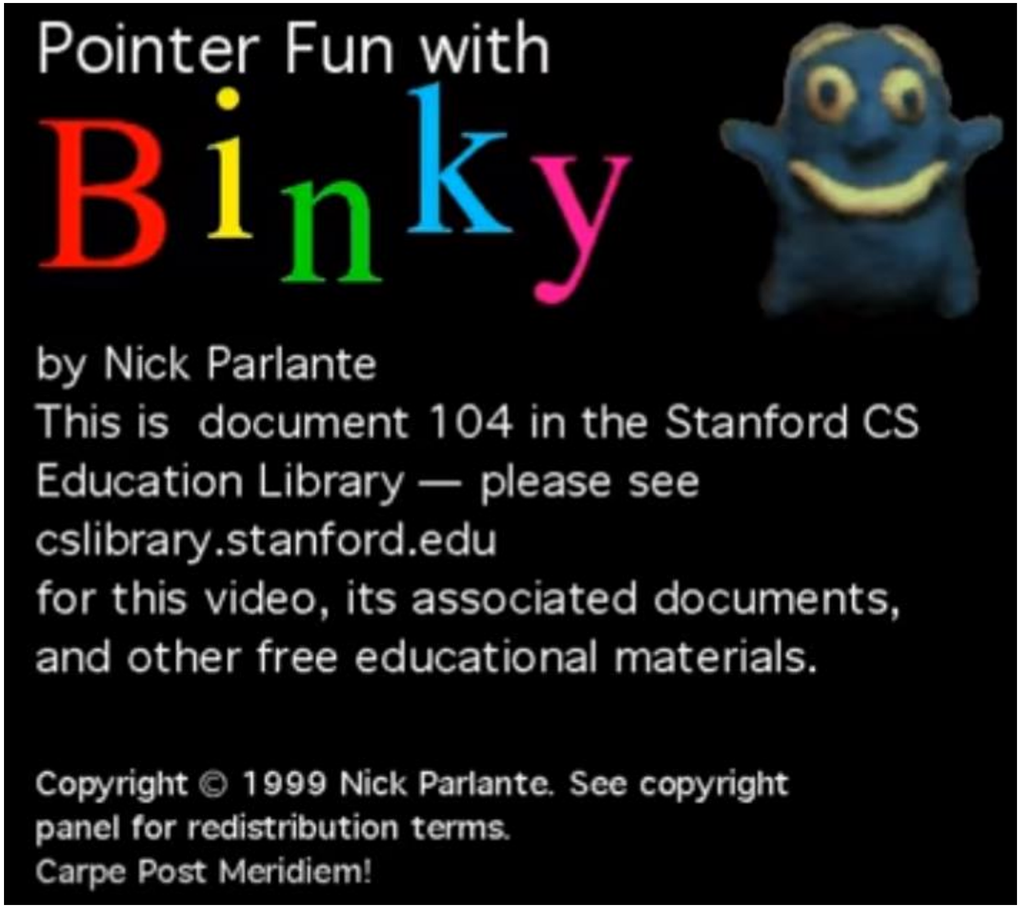
```
void foo()
{
    int a;
    Location loc;  // an object
    ...
}
```

"}" signifies send of scope. "a" and "loc" are cleared (removed from stack).

```
void foo()
{
    int a;
    Location* loc = new Location();  // an object
    ...
    delete loc;
}
```

"loc" was allocated in the heap, so we must explicitly delete it. "a" is still within the scope of the function, so it gets handled automatically in the stack.

Pointers in C/C++



https://www.youtube.com/watch?v=5VnDaHBi8dM

Pointers in C/C++

- **Pointer**
  - Memory address of a variable
- **Numbered memory locations**
  - Addresses used as name for variable
- **Pointers are typed and stored in a variable**
  - The variable does not store the value
  - Instead, it stores a pointer to int, double, etc
- **Example:**
  - **`double* p;`**
  - p is declared a "pointer to `double`" variable
  - p can hold pointers to variables of type double

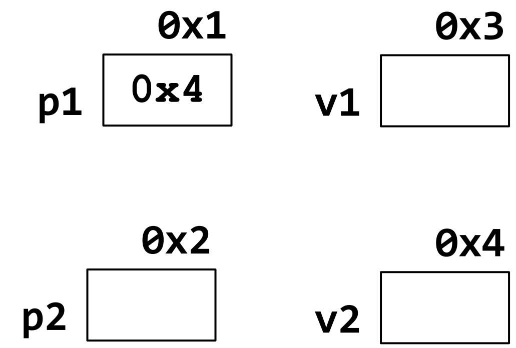
Data	Address
`01001101`	`0x107`
`01011010`	`0x106`
`00111011`	`0x105`
`00111001`	`0x104`
`10110101`	`0x103`
`11000101`	`0x102`
`01010111`	`0x101`
	`0x100`
`...`	

`double* p   vs   double *p`

Addresses and Numbers

- **Pointer is an address and address is an integer**
  - However, pointer is not an integer itself
- **C++ forces pointers be used as addresses**
  - Cannot be used as numbers
  - Even though they are numbers
- **Pointer terminology**
  - Talk of "pointing“ and not "addresses"
  - Pointer variable "points to" an ordinary variable

Pointing to …



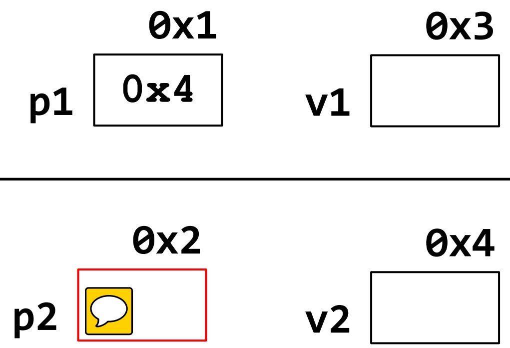
- **Example:**
  - ```
    int *p1, *p2, v1, v2;
    p1 = &v2;
    ```
  - Sets pointer variable p1 to point to int variable v1

- **Operator & determines the address of a variable**
  - Read as: “p1 equals address of v1” or "p1 points to v1”

- **Dereference operator, ***
  - Means: "Get data that p1 points to” (Binky’s dereferencing wand)

- **Two ways to refer to v1 now:**
  - Using v1 itself: `cout << v1;`
  - Via pointer p1: `cout << *p1;`
    ```

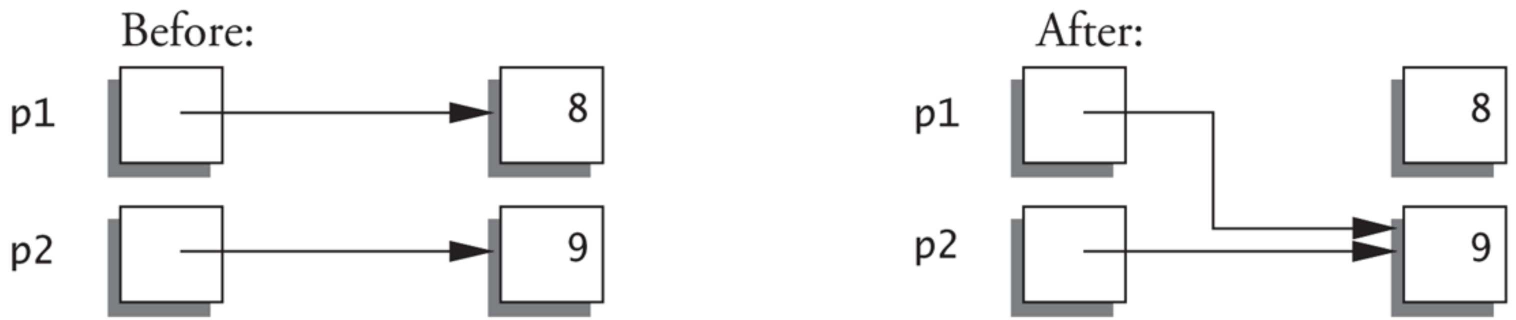
Pointer Assignments



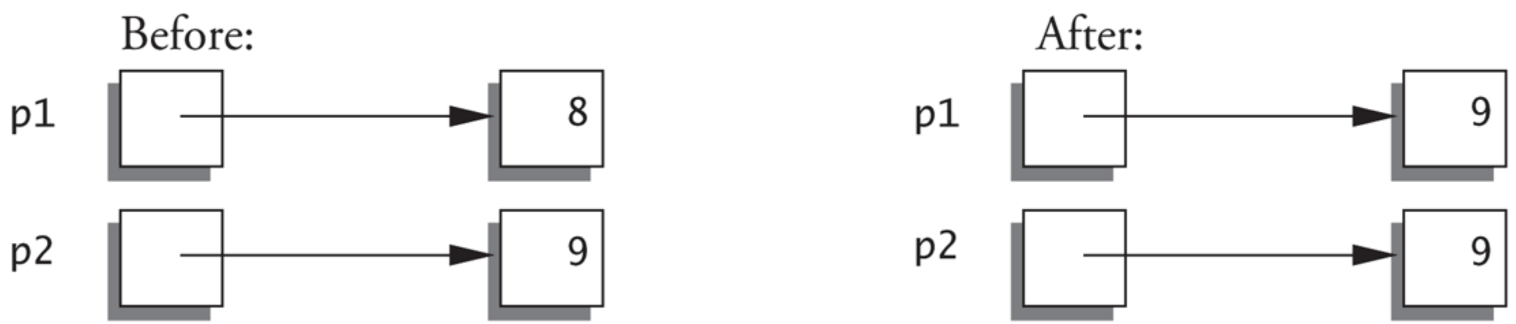
- **Pointer variables can be assigned:**
  - **`p2 = p1;`**
  - Assigns one pointer to another
  - Make p2 point to where p1 points
- **Do not confuse with: *p1 = *p2;**
  - Assigns p1 value to p2 value

Pointer Assignments Visualized

```
p1 = p2;
```




```
*p1 = *p2;
```



The new Operator

- **Can dynamically allocate variables for pointers**
  - Operator `new` creates variables (in heap)
  - Operating system finds a suitable space in the heap, and returns the (starting) location in memory that you can use.

- `p1 = new int;`
  - Allocates space in the heap to store a single integer (4 bytes)
  - Assigns p1 to point to it
  - Can access with *p1 and use it just like ordinary variable
  - **Need to clear the memory when you no longer need it to prevent memory leaks.**

Basic Pointer Manipulations /1

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
```

Basic Pointer Manipulations /2

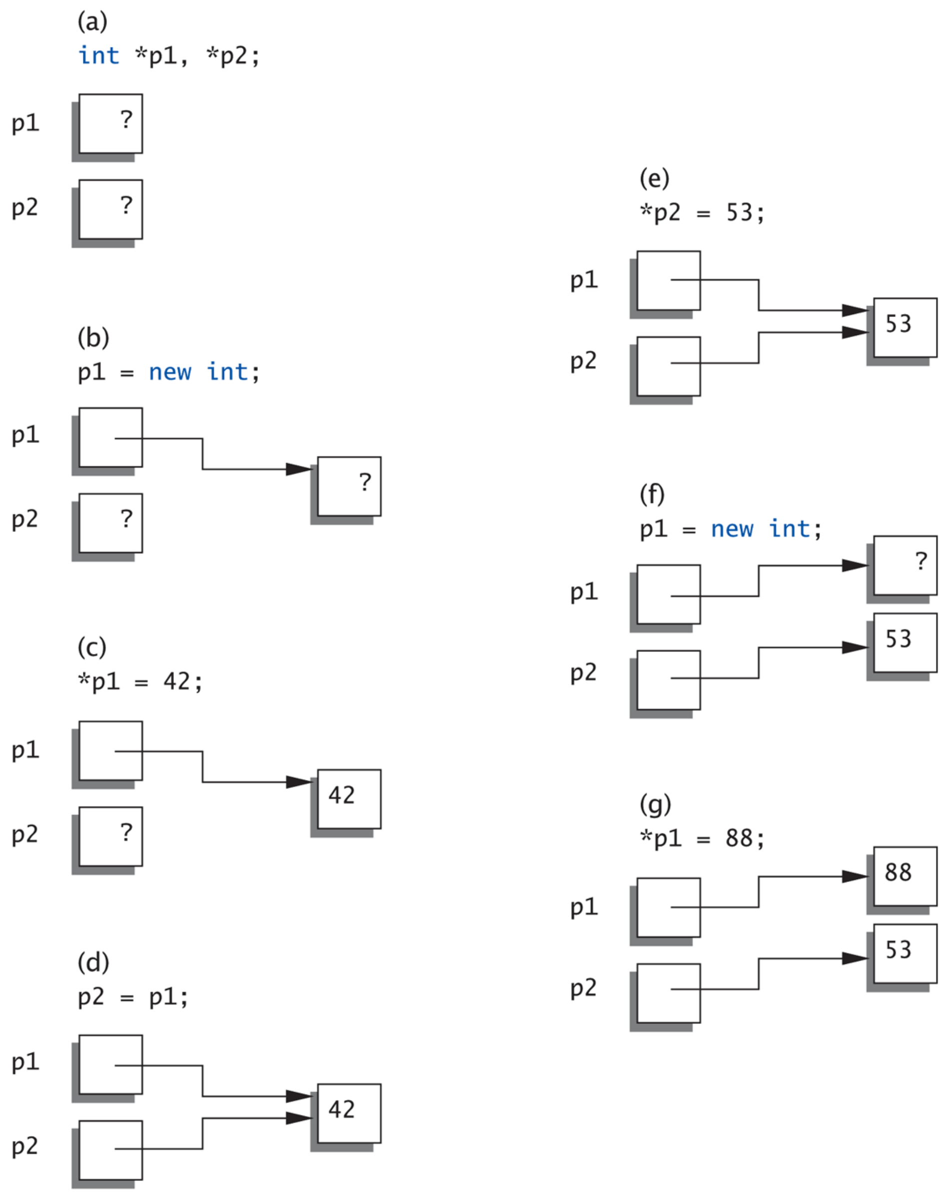
```
        p1 = new int;
        *p1 = 88;
        cout << "*p1 == " << *p1 << endl;
        cout << "*p2 == " << *p2 << endl;


        cout << "Hope you got the point of this example!\n";
        return 0;
    }
```

SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Basic Pointer Manipulations /3



More on `new` Operator

- **Allocates memory for new dynamic variable**
  - Returns pointer to the memory (heap)

- **If type is class type:**
  - Constructor is called for new object
  - Can invoke different constructor with different arguments:

```
MyClass *mcPtr;  // no constructor called yet
mcPtr = new MyClass(32.0, 17);
```

- **Can still initialize non-class types:**
  - ```
    int *n;
    n = new int(17);    //Initializes *n to 17
    ```
    ```

Pointers and Functions

- **Pointers are full-fledged C++ types**
  - Can be used just like other types
  - Can be function parameters
  - Can be returned from functions
- **Example:**
  - `int* findOtherPointer(int* p);`
  - This function declaration has "pointer to an int" parameter and it returns "pointer to an int" variable

Checking new Success

- **Test if new succeeded using try/catch block:**

```
try {
    string *p = new string("abc");
} catch (bad_alloc& ba) {
    cerr << "Bad memory allocation: " << ba.what() << endl;
}
```

- **Test if new succeeded using (nothrow) option:**

```
string *p = new (nothrow) string("abc");
if (!p) { // check if P is NULL
    cerr << "Bad memory allocation occurred" << endl;
}
```

`delete` Operator

- **Deallocate dynamic memory when not needed**
  - Returns memory to freestore
  - Example:
    ```
    int *p;
    p = new int(5);
    … //Some processing…
    delete p;
    ```
  - Deallocates dynamic memory "pointed to by pointer p"

Is “delete” the best word?

Dangling Pointers

- **`delete p;`**
  - Destroys dynamic memory
  - But p still points there; called "dangling pointer"
  - If p is then dereferenced using (*p), this leads to unpredictable results

- **Avoid dangling pointers**
  - Assign pointer that may be used again to NULL after delete:
  - ```
    delete p;
    p = NULL;
    ```
    ```

Define Pointer Types

- **Can name pointer types**
  - To be able to declare pointers like other variables
  - Eliminate need for "*" in pointer declaration

- **Use: `typedef int* IntPtr;`**
  - Defines a new type alias
  - Consider these declarations:
    ```
    IntPtr p;
    ```
    OR
    ```
    int* p;
    ```
  - The two declarations above are equivalent

Call-by-value Pointers Example /1

```
//Program to demonstrate the way call-by-value parameters
//behave with pointer arguments.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main( )
{
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "
         << *p << endl;
```

Call-by-value Pointers Example /2

```
    sneaky(p);

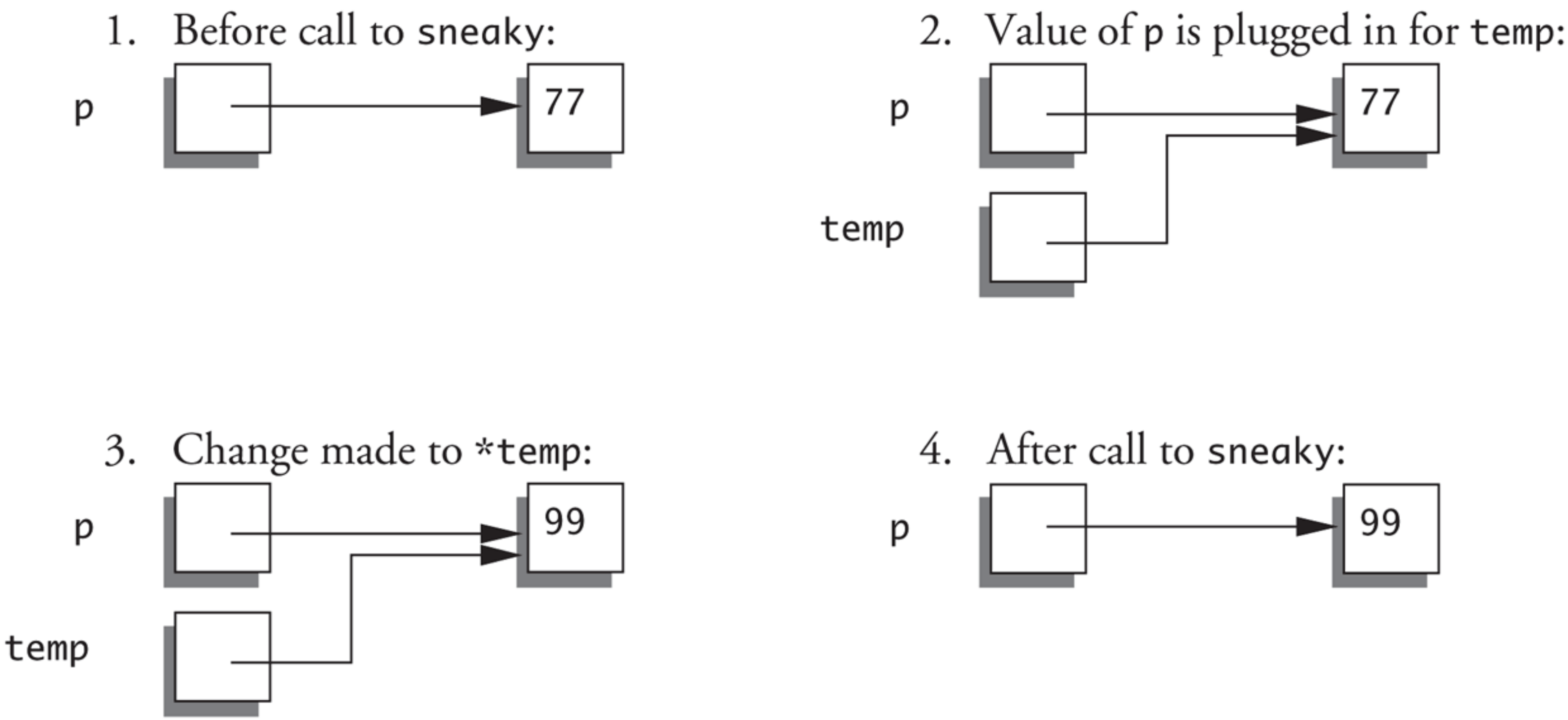
    cout << "After call to function *p == "
         << *p << endl;

    return 0;
}
void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
         << *temp << endl;
}
```

Sample Dialogue

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Call-by-value Pointers Example /3



Copy Constructors Revisited

- **Shallow Copy:**
  - Copies only the member variables
  - Also known as member-wise copy
  - Default for the assignment and copy constructors

- **Deep Copy:**
  - Pointers and dynamic memory are involved
  - Hence, must dereference pointer variables to get to data that needs to be copied; compiler will not do this for you
  - Write your own assignment overload and copy constructor in this case

Destructors Revisited

- **Dynamically-allocated variables need to be deleted**
  - Private member data variables if defined as pointers have memory assigned to them dynamically
  - For instance, data is allocated in a constructor
  - Must have means to deallocate memory when object is destroyed
  - Write your own destructor that deletes/frees the dynamically allocated memory

Classes Revisited /1

- **The -> operator: represents shorthand notation**
  - Combines dereference * operator and dot . operators
  - Specifies the member of class pointed to by a pointer

- **Example:**
  - ```
    MyClass *p;
    p = new MyClass;
    p->grade = "A";
    ```
    ```
    OR
    (*p).grade = "A";
    ```
  - The two assignments immediately above are equivalent
    ```

Classes Revisited /2

- **Member methods might need to refer to the calling object**
  - Use predefined “this” pointer if needed
  - Automatically points to the calling object:

```
Class Simple {
public:
        void showStuff() const;
private:
        int stuff;
};
```

  - Two ways for member methods to access:

```
cout << stuff; // preferred method of use

// use *this if there are multiple stuff variables
// such as, stuff as a local variable and stuff member
cout << this->stuff;
```

  - To return the object itself, use the following

```
return *this;
```

Classes Revisited /3

- **Assignment operator for dynamic allocation — example:**

```
StringClass& StringClass::operator=(
        const StringClass& rtSide) {
        // if the right side is same as the left side
        if (this == &rtSide)
                return *this;

        else {
                capacity = rtSide.length;
                length = rtSide.length;
                // free up old memory
                delete [] a;
                a = new char[capacity];
                for (int i = 0; i < length; i++)
                        a[i] = rtSide.a[i];
                return *this;
        }
}
```

Array Variables as Pointers /1

- **Standard array is of fixed size**
  - Dynamic array's size not specified at programming time
  - It is instead determined while the program running

- **Array variable refers to first indexed variable**
  - Hence, array variable is a pointer variable

- **Example:**
  - ```
    int a[10];
    typedef int* IntPtr;
    IntPtr p;
    ```
  - a and p are pointer variables
  - `p = a; // valid;  p points to the first item in a`
  - `a = p; // invalid; a is a const ptr`
    ```

Array Variables as Pointers /2

- **Array variable: `int a[10];`**
  - Interpreted as "const int *" type
  - Array was allocated in memory already
  - Variable “a” must point to the allocated memory and it cannot be changed
  - In contrast to ordinary pointers that can change

- **Static array limitations**
  - Must specify size first but the size may not be known until the program runs
  - Typically wastes memory space

- **Dynamic arrays**
  - Can grow and shrink as needed using dynamic memory

Creating Dynamic Arrays

- **Use the "new" operator**
  - Dynamically allocate them with a pointer variable and then treat them like standard arrays

- **Example:**
  - ```
    typedef double * DoublePtr;
    DoublePtr d;
    d = new double[size]; // size was already defined
    ```
  - Creates dynamically allocated array variable d with the number of elements defined by the size variable
    ```

Deleting Dynamic Arrays

- **Dynamic arrays are allocated at run-time**
  - So they should be destroyed at run-time too
- **Recall Example:**
  - ```
    d = new double[size];
    … // processing
    delete [] d;
    ```
  - De-allocates all memory for the dynamic array
  - Brackets indicate that an "array" is there
- **After deleting d’s memory, d still points there**
  - If d may be used again, should set “d = NULL;” to avoid dangling pointers
    ```

Function that Returns an Array

- **Array type is not allowed as return-type of function**

- **Example:**
  - **`int[] someFunction();`** `// INVALID!`
  - Correction: return pointer to array base type
  - **`int* someFunction();`** `// VALID!`

- **Practical considerations:**
  - Use **`std::vector`** instead of primitive arrays, even the dynamically created ones, for performance reasons
  - If using static arrays, use **`<array>`** (C++11) instead

  - **`<array>`** provides reflexivity of the array itself, including the at() function that allows you to safely query the array
  - The at(index) function throws an **`out_of_range`** exception if the specified index is out of the array range

Pointer Arithmetic

- **Recall: One can perform arithmetic on pointers**
  - However, only addition and subtraction using integer values work on pointers

- **Example:**
  - ```
    typedef double* DoublePtr;
    DoublePtr d;
    d = new double[10];
    ```
  - d contains address of d[0]; d + 1 evaluates to address of d[1]; d + 2 evaluates to address of d[2]

  - Iterate through the array without indexing:
    ```
    for (int i = 0; i < arraySize; i++)
      cout << *(d + i) << " " ;  // for demonstration
    ```
  - Equivalent to:
    ```
    for (int i = 0; i < arraySize; i++)
      cout << d[i] << " " ;  // preferred way
    ```
    ```

Multidimensional Dynamic Arrays

- **Recall: Arrays of arrays**
  - ```
    typedef int* IntArrayPtr;
    IntArrayPtr *m = new IntArrayPtr[3];
    ```
  - The above creates an array of three pointers
  - ```
    for (int i = 0; i < 3; i++)
          m[i] = new int[4];
    ```
  - Results in three-by-four dynamic array

Final Topic: Class Separation

- **Program parts are kept in separate files**
  - Compiled separately
  - Linked together before the program runs

- **Class Independence**
  - Independent class specification called class interface
  - Separated from the class implementation in another file
  - If the implementation changes, only the implementation file needs to be changed
  - Class specification does not need to change
  - Hence, the programs using the class do not need to change as the result of the class change

Class Interface Files

- **Interface File**
  - Contains class definition with function and operator declarations/prototypes
- **Class interface always in header file**
  - Use .hpp or .h naming convention
- **Programs that use the class will include it**
  - Use **`#include "myclass.hpp"`** in your .cpp code
  - Quotes indicate that it is a local header, and that it can be found in your working directory
  - **`#include`** is basically copy+paste
- **< > indicate predefined library header file**
  - To be found in the library directory (e.g., **#include <iostream>**)

Using #ifndef in Class Interface Files

- **Header file structure:**
  - ```
    #ifndef FNAME_HPP
    #define FNAME_HPP
    … // contents of the header file
    …
    #endif
    ```
  - FNAME typically name of file for consistency and readability
  - This syntax avoids multiple definitions of header file
    ```

Class Implementation Files

- **Class implementation in .cpp file**
  - Typically give interface file and implementation file same name, such as myclass.hpp and myclass.cpp
  - All class member functions are defined in this file
  - Implementation file must #include class header file

- **.cpp files typically contain executable code**
  - Function definitions go into the class implementation file
  - The main() method and other program-specific code is typically located in the program file (e.g., myprogram.cpp)

Aside: Smart Pointers

- As you can probably tell, managing dynamic memory via pointers can get really tricky with larger programs.
- There have been initiatives to create "smart pointer" types that "automatically" delete (free) the memory when it's out of scope.
  - Combines dynamic and local variable concepts
- Google has a good set of smart pointer types in their style guide:

https://google.github.io/styleguide/cppguide.html#Ownership_and_Smart_Pointers

https://www.chromium.org/developers/smart-pointer-guidelines

Objectives

Core Content:

- Introduction to Virtual Address Space
- C++ Memory Model
- Classes and Dynamic Memory
- Dynamic Arrays
- Separate Compilation Units

Additional Information:

- **Namespaces via `using` Directives**
- **Global and Unnamed Namespaces**
- **Nested Namespaces**

Action Items

- **Read:**
  - Chapter 2 (Linked Data Structures) from the course handbook
- **Additional Readings:**
  - Chapter 4 from “Data Structures and Other Objects Using C++” by Main and Savitch
  - Review Chapters 9 – 11 from “Absolute C++” by Savitch
    - Review the material discussed above in more detail

Namespaces

- **Namespace:**
  - A collection of name definitions
  - That is, class definitions and variable declarations

- **Programs use many classes, functions**
  - Commonly these can have the same names
  - Namespaces help resolve name conflicts

- **Examples:**
  - Person::firstName
  - Person::testAge
  - CS246Namespace::A5

Namespace `std`

- **Std namespace contains all names defined in many standard library files**

- **Example: #include <iostream>**
  - Places all name definitions (cin, cout, etc.) into std namespace
  - Must specify this namespace for program to access names

- **using namespace std;**
  - Makes all definitions in std namespace available
  - If one needs to redefine cout and cin, this statement should not be used
  - Reference cout and cin from std directly as **`std::cout and std::cin`**

Global Namespace

- **All code goes in some namespace**
  - Unless specified, this is the global namespace
  - No need for the using directive since the global namespace always available

- **Multiple namespaces**
  - What if a name is defined in both namespace, such as global and std?
  - Name conflicts result in an error
  - Can still use both namespaces, but must specify which namespace is to be used at a particular time

Specifying Namespaces

- **Given namespaces NS1, NS2**
  - Both have void function **`myFunction()`** defined differently
  - ```
    {
          using namespace NS1;
          myFunction();
    }
    {
          using namespace NS2;
          myFunction();
    }
    ```
  - **`using`** directive has block scope
    ```

Creating a Namespace

- **Use namespace grouping:**
  - ```
    namespace Name_Space_Name
    {
          // some code
    }
    ```
  - Places all names defined in Some_Code into namespace Name_Space_Name
  - Can then be made available:
    `using namespace Name_Space_Name`
    ```

Creating a Namespace Example

- **Name your namespace with a unique string**
  - Reduces chance of other namespaces with same name

- **Function Declaration:**
  - ```
    namespace MTE140
    {
            void greeting();
    }
    ```

- **Function Definition:**
  - ```
    namespace MTE140{
        void greeting()  {
            cout << "Hello from MTE140“ << endl;
        }
    }
    ```

using Declarations

- **Can specify individual names from namespace**

- **Example:**
  - Namespaces NS1 and NS2 exist
  - Each has functions **`fun1()`** and **`fun2()`**

  - Declaration syntax:
    `using Name_Space::One_Name;`

  - Specify which name from each namespace:
    ```
    using NS1::fun1;
    using NS2::fun2;
    ```

Qualifying Names

- **Can specify where the name comes from**
  - Use qualifier and scope-resolution operator
- **Example:**
  - ```
    NS1::fun1(); // specifies that fun()
                 // comes from namespace NS1
    ```
- **Especially useful for parameters:**
  - `int getInput(std::istream inputStream);`
  - Parameter found in istream's std namespace
  - Eliminates need for the using directive
    ```

Unnamed Namespaces

- **Compilation Unit Defined:**
  - A file along with all the files #included in it

- **Every compilation unit has unnamed namespace**
  - Written the same way but with no name
  - All names are then local to the compilation unit
  - Use unnamed namespace to keep things local
  - Scope of unnamed namespace is the compilation unit

- **Global vs. Unnamed Namespaces**
  - Global namespace: No namespace grouping; global scope
  - Unnamed namespace: Has namespace grouping, just no name; local scope

Nested Namespaces

- **Legal to nest namespaces**
  - ```
    namespace S1
    {
    namespace S2
    {
            void sample()
            {
                    …
            }
    }
    ```

- **Qualify names twice:**
  - `S1::S2::sample();`
    ```

Hiding Helper Functions

- **Helper function:**
  - Typically a low-level utility function and not for public use

- **Two ways to hide helper function from public use:**
  - Place the function in the unnamed namespace if the function does not require access to the internals of the object
  - Make private member function if the function requires access to the internals of the object