void f()
{
    // x, y are local variables
    // on the runtime stack
    int x = 4;
    Word y("Brokeback");

    // p is another local variable
    // on the runtime stack.
    // But the array of 100 int
    // that p points to
    // is on the heap
    int* p = new int[100];
}
Memory Usage on Runtime Stack and Heap

• **Local** variables are *constructed* (created) when they are defined in a function/block on the **run-time stack**.

• When the function/block terminates, the local variables inside and the CBV arguments will be *destructed* (and removed) from the run-time stack.

• Both construction and destruction of variables are done automatically by the compiler by calling the appropriate **constructors** and **destructors**.

• **BUT**, dynamically allocated memory remains after function/block terminates, and it is the user’s responsibility to return it back to the **heap** for recycling; otherwise, it will stay until the program finishes.
Garbage and Memory Leaks

```c
int main()
{
    for ( int j = 1; j <= 10000; ++j )
    {
        int* snoopy = new int[100];
        int* vampire = new int[100];
        snoopy = vampire;  // Now snoopy becomes vampire
        .....               // Where is the old snoopy?
    }
}
```

- **Garbage** is a piece of storage that was created (allocated) by a program, where there are no more pointers/references to it.
- A **memory leak** occurs when there is garbage.

Question: What happens if there is a huge piece of garbage, or garbage is continuously created inside a big loop?!
Example: Before and After \( p = q \)

delete: to prevent garbage

int main()
{
    Stack* p = new Stack(9);  // A dynamically allocated stack object
    int* q = new int[100];     // A dynamically allocated array of integers
...
    delete p;                 // delete an object
    delete [] q;              // delete an array of objects
    p = NULL;                 // it is good practice to set a pointer to 0
    q = NULL;                 // when it is not pointing to anything
}

• Explicitly deallocate the memory for a single object by calling delete on a pointer to the object.
• Explicitly deallocate the memory for an array of garbage objects by calling delete [] on a pointer to the first object of the array.
• Notice that delete ONLY puts the dynamically allocated memory back to the heap, and the local variables (p and q above) stay behind on the run-time stack until the function terminates.
Dangling References and Pointers

However, careless use of `delete` may cause dangling references.

```c
int main()
{
    char* p;
    char* q = new char [128]; // dynamically allocate a char buffer
    ...
    p = q;                    // p and q now points to the same char buffer
    delete [] q; q = 0;       // delete the char buffer
    // Now p is a DANGLING POINTER!
    p[0] = 'a';               // Error: possibly segmentation fault
    delete [] p;             // Error: possibly segmentation fault
}
```

- A dangling reference is created when memory pointed to by a pointer is deleted but the user thinks that the address is still valid.
- Dangling references are due to carelessness and pointer aliasing — where an object is pointed to by more than one pointer.
Example: Dangling References

BEFORE

p: 0x8a48
q: 0x8a48

AFTER

delete [ ] q; q = 0;

p: 0x8a48
q: 0
Other Solutions: Garbage, Dangling References

Memory leaks and dangling references are due to careless pointer manipulation, especially in situations where there is pointer aliasing.

– Some languages provide automatic garbage collection facility which stops a program from running from time to time, checks for garbage, and puts that memory back in the heap for recycling.
  • e.g.: Lisp, Scheme, Java, C#, .NET …

– Some languages do not have explicit pointers at all!
  ( The large majority of program bugs are due to pointers. )

– However, you pay a performance penalty for such solutions.
Destructors: Introduction

```c
void Example()
{
    Word x( "bug", 4 );
    ...
}

int main() { Example(); ... }
```

- On return from `Example()`, the local `Word` object `x` of `Example()` is destroyed from the run-time stack of `Example()`. i.e. the memory space of `(int) x.frequency` and `(char*) x.str` are released.

Quiz: How about the dynamically allocated memory for the string, “bug” that `x.str` points to?
Destructors

C++ supports a more general mechanism for user-defined destruction of class objects through destructor member functions.

```cpp
~Word() { delete [] str;}
```

- A destructor of a class X is a special member function with the name X::~X().
- A destructor takes no arguments, and has no return type – thus, there can only be ONE destructor for a class.
- The destructor of a class is invoked automatically whenever its object goes out of scope – out of a function/block.
- If not defined, the compiler will generate a default destructor of the form X::~X(){ } which does nothing.
Example: Destructors

class Word {
    int frequency;
    char* str;

public:
    Word(): frequency(0), str(0) { }
    Word(const char* s, int k = 0) { ... }
    ~Word() { delete [] str; }
}

int main() {
    Word* p = new Word("Brokeback Mountain");
    Word* x = new Word [5];
    ...
    delete p;     // destroy a single object
    delete [] x;  // destroy an array of objects
}
Bug: Default Assignment

```c
void buggy(Word& x)
{
    Word bug("bug", 4);
    x = bug;
}

int main()
{
    Word movie("Brokeback Mountain"); // which constructor?
    buggy(movie);
}
```

Quiz: What is `movie.str` after returning from the call `buggy(movie)`?