Dynamic memory.

Have patience with pointers: You really need them to use C++
and how to swap variables' values.

First, review reference parameters.

Dynamic Memory via new

Pointers and

See §4.1 of DS

CSI 310: Lecture 4
called function
but are READ-ONLY in the
Const. Reference parameters ditto,
Which of the XXXX’s is made 98??
RP = 98; Changes CV
the caller's argument variable
Reference Parameters REFER TO

The caller’s argument variable
(2) called function must MODIFY
and/or
(1) sizeof(parameter) is big
Reference Parameters are great if

Any value parameter

Each VALUE parm...inside called fun’s body...}

const ref. parm.  //value parameter.

const ref. parm. //reference parameter.

void fun ( int VP,

//value parameter.

int RP,  //reference parm.

//const ref. parm.

const int& RP,  //reference parm.

//value parameter.

void caller()

//inside called fun’s body...

Changes CV

Which of the XXXX’s is made 98??

//Called function.

//caller's variable.

RP = 98;

//fun (CV, CV, CV);
reference, int & from variable, int

Warning: initialization of non-const

& prints the warning.

argument values, and the reference parameters refer to THBM.

Answer: Temporary variables are created, initialized to the

{ RP1 = 0; RP2 = 0; }

void fun(inta RP1, inta RP2)

where fun() is

fun(38, x+39);

We call

a reference parameter?

What if a constant or a computed value is used as an argument for

An annoying detail:
must NOT PRINT ANY WARNINGS.

Rule for CSCI 310: All compilations must be done with -Wall
and

(iii) (not 39) (bad)

Program compiles and might print

You get a warning about a conversion that discards const; BUT the

`count >> TINNE` `end`;

`fun(TINNE, TINNE)`;

`const int TINNE = 39`;

What if? `void caller()`
TEMPE

Try first \texttt{j = i}; or first \texttt{i = j}; \textquote{That cannot work!!!}

How can you program swapping the values of \texttt{i} and \texttt{j}?

\texttt{j = 2003};
\texttt{i = 1928};
\texttt{\texttt{\textquote{That}}}
Currently in each of the 3 variables, before and after each step…

Please simulate the execution of this by writing the values:

TEMP = I
I = J
J = TEMP

Garbage!

?Junk!
Worry about this yet...

sizes of variables are comprised of different numbers of bytes. Don't
One big house might cover several adjacent lots. Technically, different

DIFFERENT FROM the variable's current VALUE.

addresses. The address of a memory location or byte IS VERY
locations (called bytes). The byte numbers are called memory

A segment of computer memory consists of numbered memory

DIFFERENT FROM the house's current OCCUPANTS.

numbers are called addresses. The address of a house IS VERRY

A block of real estate consists of numbered houses. The house

( on variables).

A pointer is the memory address of a variable. (Review Lecture)
The value of this (unnamed) variable is 57.

Pointer value or address points to a 4-byte integer variable.

57
The pointer variable whose address is 987 is pointing to the above integer variable whose address is 992 and value is 57.

Here’s a pointer variable whose address is 987. Its address is 57. The integer variable with value 57 is the address of the above integer variable whose address is 992 and value is 992.

I will denote pointer values by black dots because the numeric value is usually boring.

Real programmers write their addresses in hexadecimal (base sixteen) because hex to binary (base 2) conversion is very easy.

(You will see hex in Lab2.)

You can store in any other variables, like numbers, and they can be stored in (pointer) variables.

Here’s the integer variable whose VALUE (992) is the address of the above integer variable whose address is 992 and value is 57.
THE VALUE of $x$ is 98.6.

THE ADDRESS of $x$ is 0x00000000.

------------

```cpp
{ cout << "THE VALUE of $x$ is " << x >> end;
 cout << "THE ADDRESS of $x$ is " << &x >> end;
 float x = 98.6;
 }

main()

using namespace std;

#include <iostream>

operand, which must be a variable.
The C/C++ @ operator provides the ADDRESS OF its
An array element is a variable, so it produces its address:

```c
float x[3] = {-37.6, 15.3, 98.6};
```
```cpp
try {
    ckx: In function `main()
    $ g + - e

{ }

cout << x(x+3) >> end!
    `int x = 3;
} 
main

using namespace std;

#include <iostream>

Expressions, like (x+3), are not variables, so:
```
but $x$ equals 98.6
myStor{pointer equals 0x{p[i|j\]60
0x{p[i|j\]60
$x$ equals
}$a$ out

{cout $\gg$ "$x$ equals"
cout $\gg$ "myStor{pointer equals "$x$ equal{cous $\gg$ $x$ equals
myStor{pointer $=$ $x$;
float $x$ $=$ 98.6;
} float *myStor{pointer;

A POINTER VARIABLE can store a pointer value.
The type of a variable indicates the kind of data it can store.

The declaration of a pointer variable specifies BOTH:

1) that it IS a pointer variable, and
2) the type of those variables it can point to.

The name of the newly declared pointer variable can hold the address of the type of variable it can point to.

The type of a variable indicates the kind of data it can store.

\*MyFirstPointer

\*float

The name of the newly declared pointer variable can hold the address of the type of variable it can point to.

Learn how pointer variables are declared in C++.
NULL. This is useful. You’ll see.

Often, a pointer variable stores the illegal address 0, which is called

(usually, a pointer variable can store its own address).

```
OTHER variable

 some data type

 DATA

 an address

 VartableName
```

answer: Addresses of (usually) OTHER variable

```
some data type * VartableName;
```

Remember a pointer variable is declared by

```
What kind of data can a pointer variable store?
```
MyP does NOT point to 2003.

Now we can say "MyP points to MyInt."

int MyInt = 2003;

MyP = &MyInt;

Computer memory:

produces in the

The value of MyP is the address of the int variable MyInt

Means

Pointer variable MyP points to int variable MyInt
times code that.

* can also dereference a pointer constant. Device driver programmers some-

```cpp
0xBE0FE04

cout >> Myp;

2003

cout >> *Myp;

2003

cout >> MypInt;
```

That means: Access the variable whose address is in the pointer.

The * operator dereferences a pointer variable.
Before Multiplication:

\[ \text{MyInt} = \text{MyP} \times 2 \]

\[ \text{MyP} = \text{MyInt} \]

\[ \text{MyInt} = 2003 \]

\[ \text{MyP} = \text{MyInt} \]

\[ \text{MyInt} = 4006 \]

After Multiplication:

\[ \text{MyP} = \text{MyInt} \]

\[ \text{MyInt} = \text{MyP} \]

\[ \text{MyInt} = 4006 \]

\[ \text{MyP} = \text{MyInt} \]

\[ \text{MyInt} = 4006 \]
This is a "CRASH". Computer tried to read memory at the illegal Segmentation Fault.

```cpp
int main() {
    MyP = 0x0;
    MyInt = 0x0;

    cout << "MyP = " << MyP; // 0x0
    cout << " MyInt = " << MyInt; // 0x0

    MyP = NULL; // NULL
    cout << " MyP = " << MyP; // NULL
    cout << " MyInt = " << MyInt; // NULL
}
```

Before Assignment:

![Before Assignment Diagram]

After Assignment:

![After Assignment Diagram]
currently in each of the 3 variables, before and after each step.

Please simulate the execution of this by writing the values

\[
\begin{align*}
pt_2 &= \text{TEMP}  \\
pt_1 &= pt_2  \\
\text{TEMP} &= pt_1  \\
\text{int* pt}_1 &= \text{int* pt}_2  \\
\text{pointers?}
\end{align*}
\]

How can you program swapping the values of \( I \) and \( J \) using

\[
\begin{align*}
I &= 1928  \\
J &= 2003  \\
\text{int } I, J;
\end{align*}
\]
assert (p1 == q1) && (p2 == q2);

1928 2003

cout >> I >> J >> endl;

2003 1928

cout >> *p1 >> *p2 >> endl;

p2 = TEMPp;

p1 = p2;

TEMPp = p1;

int *TEMPp = p1;

p1 = q1;  p2 = q2;

int *p1 = int *p2;

int I;  int J;

int J;  int I;

1928;  J = 2003;

different?

Now, how is swapping the values of the pointer variables
The C/C++ compiler uses this formula for you. You won’t need it.

Formula: \[ a[I] = base \times size + I \]

So given (1) the address of element 0, base, and (2) the size of each entry (number of bytes), and (3) the element index (value of I), the computer can quickly compute the address of A[I].

We code access to the ith entry with A[I]. It is a variable, composed of a sequence of variables called entries.

What is an array?

\[ [4] [3] [2] [1] [0] \]
prints

\[ p_1 \]
\[ p_1[0] \]
\[ p_1[1] \]
\[ p_1[2] \]
\[ p_1[3] \]
\[ p_1[4] \]

\$7$
\$9$
\$11$
\$13$

\$p_1[3] = p_1[2] \$
\$p_1[2] = p_1[1] \$
\$p_1[1] = p_1[0] \$
\$count >> p_1[0] >> endl \$
\$p_1[1] = 7 \$
\$p_1 = \text{the address of some integer array} \$
\$\text{int } *p_1; \$
\$p_1 \$
\$e \$
\$e \$
\$e \$
\$e \$
\$p_1 \$
\$e \$
\$e \$
\$e \$
\$e \$
\$e \$
\$e \$

If \( p_1 \) is a \text{int} pointer variable whose value is the address of an \text{int} array, then \( p_1[0] \) is \text{array access entity 0}, \( p_1[1] \) is \text{array access entity 1}, etc.

\text{In C/C++}, \text{dereferencing a pointer \( p_1 \) using array notation:}
cout << PART[3].FLOW() << endl;

PART[1].SHITT(2);

PART[0].SHITT(1); equivalent to (*PART).SHITT(1);

PART = the address of some array of throttles;

throttle *PART;

This works for an array of ANYTHING, e.g.
and object construction/destruction.

Sophisticated SW designs distinguish between memory allocation/recycling.

The program runs.

explicitly coded operation (new and delete) executed when

But, a dynamic variable is created or destroyed only by an
called and are destroyed (storage recycled) when it returns.

3. Automatic (local) variables are created when a function is

dynamic variables are created.

Unpredictable input data can determine if and how many

2. Dynamic variables are created during process execution.

identifiers. But they, like all variables, have memory addresses.

1. Dynamic variables are not declared. They are not named by

(You need pointers to access them!)

Dynamically Allocated Variables.
new is a keyword in C++. See Figure 4.1.

```c
PAT = new throttle[5];
throttle*PAT;
//To allocate an array of 5 throttles:
PI = new int[4];
array form for new
//To allocate an array of say 4 ints, do the same except use the

PI = new int;
and then make program execute:
e.g.:int*PI;

a pointer variable to hold its address,

To allocate ONE variable of any type, say int, make sure you have

Doing dynamic allocation: use new.
```
algorithm runs new to allocate a BIGGER array. When it fills up, the insertion
allocated partially filled array. The number of items in a
practically unlimited. The implementation is a dynamically

DSO chapter 4 improves this bag class, so the number of items is
array of items.

The implementation data structure is a fixed size partially filled
array up to a fixed number of items (counting multiplicities).

DSO chapter 3 teaches a primitive bag class, for which each bag
Roadmap:
Invariant for the Reiesed Bag Class

```

class Bag {
    size_t capacity; // Current capacity of the bag.
    size_t used; // How much of the array is used.
    value_type *data; // Pointer to dynamic array.
    private:
        ... public: }
```
Or, is it the value of a pointer variable, that is, an address?

variable.

So, when you or others say "pointer", think hard: Is it a pointer?

But most everyone, we and DSO, say, for short, "PVAR is a

(int the NULL value, or else some illegal value.

store an address of a C/C++ int variable, or else it might have

name of a pointer type variable. The variable named PVAR might

we and DSO said "a pointer is an address". PVAR is really (the

int *PVAR; What is PVAR??? Is it a "pointer"??

int integer.

variable whose type is int. Thus variable stores a C/C++

an (the name of an)

Most say "It's an integer!" but, really, IVAR is (the name of an)

int IVAR; What is IVAR??

A linguistic pitfall—try not to fall into it!
Technically, “pointer” and “int” describe C/C++ types. Each variable has a type which determines what values it can hold and what operations can be done.

Perhaps we should always use the word “address” for “pointer value”.