Introduction to C++

A Basic C++ Program

```
#include <iostream>
#include <math.h>

using namespace std;

int main()
{
   float x;

   cout << "Enter a real number: " << endl;
   cin >> x;

   cout << "The square root of " << x << " is: "
              << sqrt(x) << endl;
}
```


A Basic C++ Program

```
// another C++ program
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main(){
   int a=23;
   int b=34;

   cout << "Enter two integers:" << endl;
   cin >> a >> b;
   cout << endl;

   cout << "a + b =" << a+b << endl;
   return 0;
}
```


Data Types

- **C++** is a **strongly typed** programming language where every variable has a type, name, value, and location in memory
- The **type** of a variable defines the contents of the variable. Every **type** is either:
  - Primitive
  - User-defined

Primitive Data Types

There are six common primitive types in C++:

- **`int`** - integer: a whole number.
- **`char`** - a single character/single byte
- **`bool-`** stores a Boolean (true or false)
- **`float`** - floating point number: i.e. a number with a fractional part.
- **`double`** - a double-precision floating point value.
- **`void`** - valueless special purpose type

User-defined Types

An unbounded number of user-defined types can exist – we'll create many of our own!

Two very common user-defined types:

- **std::string**, a string (sequence of characters)
- **std::vector**, a dynamically-growing array

C++ Standard Library

- The **C++ standard library** (std) provides a set of commonly used functionality and data structures to build upon.
- The C++ standard library is organized into many separate sub-libraries that can be #include'd in any C++ program
- The `iostream` header includes operations for reading/writing to files and the console itself, including **std::cout**.

7

Namespaces

- All functionality used from the standard library will be part of the **std namespace**.
- Namespaces allow us to avoid name conflicts for commonly used names.
- If a feature from a namespace is used often, it can be imported into the global space with **using**:

using std::cout;

8

Basic control structures

All C++ programs are written in terms of 3 control structures:

- Sequence structures: Built into C++. Programs executed sequentially by default.
- Selection structures: C++ has three types: **`if`**, **`if`**/**`else`**, and **`switch`**
- Repetition structures: C++ has three types: **`while`**, **`do`**/**`while`** and **`for`**

9

Exercise 1

- What is the output from the following loop?

```
for ( int i=0; i < 5 ; i++) {
   cout << i;
}
cout<<endl;
```

10

Exercise 2

- What is the output from the following loop?

```
for ( int i = 0; i < 10 ; i += 2) {
   cout << i << endl ;
}
```

11

Exercise 3

What is the output?

```
int i = 24 ;
while ( i > 0) {
   cout << i << endl ;
   i /= 2 ;
}
```

12

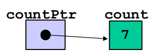
Pointers

- Normal variables contain a specific value (direct reference)

```
int count = 7;
```

- Pointer variables contain memory addresses as their values

```
int * countPtr;
countPtr = & count;
```



13

Pointer Variable Declarations and Initialization

- A pointer declaration takes the following form:
  *type *identifier;*

e.g.

```
int *myPtr;
```

  – Declares a pointer to an `int` (pointer of type `int *`)

- We can declare pointers to any data type.

e.g. `float *fptr;` `char *cptr;`

- We usually initialize pointers to `nullptr`
  - `nullptr` – points to nothing

e.g.

```
myPtr = nullptr;
```

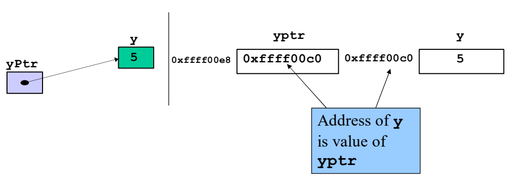
14

Pointer Operators

- `&` (address operator) - Returns the address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;       // yPtr gets address of y
```

  – **`yPtr` "points to" `y`**



15

Pointer Operators

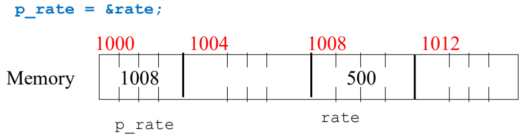
- `*` (indirection/dereferencing operator)
  – Returns an alias of what its operand points to
  – `*yptr` returns `y` (because `yptr` points to `y`)
  – `*` can be used for assignment

```
*yptr = 7;  // changes y to 7
```

- `*` and `&` are inverses
  – They cancel each other out

16

```
int rate;
int *p_rate;

rate = 500;
p_rate = &rate;
```




```
/* Print the values */
cout <<"rate = "<< rate << endl;  /* direct access */
cout <<"rate = "<< *p_rate << endl; /* indirect access */
```

17

Exercise 4

```
int a, b, *p;

a = b = 7;
p = &a;
// 1st print statement
cout << "*p = " << *p << endl;

*p = 3;
// 2nd print statement
cout << "a = " << a << endl;

p = &b;

*p = 2 * *p - a;

// 3rd print statement
cout << "b = " << b << endl;
```

18

Passing parameters to functions by value

```
void SetToZero (int var)
{
   var = 0;
}
```

- You would make the following call:

```
SetToZero(x);
```

- This function has no effect whatever to change the value of x.
- This is referred to as *call-by-value.*

Passing parameters by reference

```
void SetToZero (int *ip)
{
   *ip = 0;
}
```

- You would make the following call:

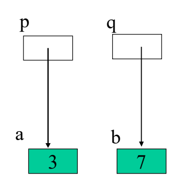
```
SetToZero(&x);
```

This is referred to as *call-by-reference using pointers.*


```
/* Swapping arguments (incorrect version) */
#include <iostream>

void swap (int p, int q)
{
      int tmp;

      tmp = p;
      p = q;
      q = tmp;
}

int main (void)
{
      int a = 3;
      int b = 7;
      cout << a << b << endl;
      swap(a,b);
      cout << a << b << endl;
      return 0;
}
```



```
/* Swapping arguments (correct version) */
#include <iostream>

void swap (int *p, int *q)
{
      int tmp;

      tmp = *p;
      *p = *q;
      *q = tmp;
}

int main (void)
{
      int a = 3;
      int b = 7;
      cout << a << b << endl;
      swap(&a, &b);
      cout << a << b << endl;
      return 0;
}
```



References

- References are a type of C++ variable that act as an *alias* to another variable.
- A reference variable acts just like the original variable it is referencing.
- References are declared by using an ampersand (&) between the reference type and the variable name.

Example

```
int n = 5, m = 6;
int &rn = n;

n = 6;
rn = 7,
cout << n << rn <<  m << endl;
rn = m ;
cout << n << rn << m << endl;
```

You cannot declare a reference without giving a value.

Another Example

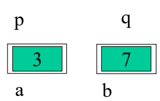
```
int * p = new int;
*p = 10;
int &r = *p;
r++;
cout << *p << endl;
```

25

```
/* Swapping arguments - with reference variables*/
#include <iostream>

void swap (int &p, int &q)
{
      int tmp;

      tmp = p;
      p = q;
      q = tmp;
}
int main (void)
{
      int a = 3;
      int b = 7;
      cout << a << b << endl;
      swap(a, b);
      cout << a << b << endl;
      return 0;
}
```

26

```
/* Swapping arguments - with reference variables*/
#include <iostream>

void swap (int &p, int &q)
{
      int tmp;

      tmp = p;
      p = q;
      q = tmp;
}
int main (void)
{
      int a = 3;
      int b = 7;
      cout << a << b << endl;
      swap(a, b);
      cout << a << b << endl;
      return 0;
}
```



27

Exercise 5

What is the output?

```
void fun1(int *a, int b){
   b = b - 1;
   *a = *a + b;
   cout << *a << "  " << b << endl;
}
int main(){
   int x=3, y=3;
   fun1(&x,y);
   cout << x << "  " << y << endl;
}
```

28

Exercise 6

What is the output?

```
void fun1(int *a, int &b){
   b = b - 1;
   *a = *a + b;
   cout << *a << "  " << b << endl;
}
int main(){
   int x=3, y=3;
   fun1(&x,y);
   cout << x << "  " << y << endl;
}
```

29

Exercise 7

What is the output?

```
void fun2(int &a, int b){
   a = a * 2;
   b = a + b;
   cout << a << "  " << b << endl;
}
int main(){
   int x=3, y=5;
   fun2(x,y);
   cout << x << "  " << y << endl;
}
```

30

Classes and Objects

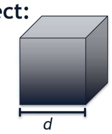
- **Class:** a type definition that includes both
  - data properties, and
  - operations permitted on that data
- **Object:** a variable that
  - is declared to be of some Class
  - therefore includes both data and operations for that data
- **Appropriate usage:**
  "A variable is an instance of a type."
  "An object is an instance of a class."

31

C++ Classes

C++ classes encapsulate data and associated functionality into an **object**:

Object:



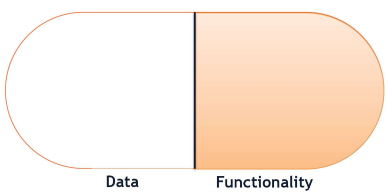
C++ class:

```
class Cube {
  public:
    double getVolume();
    // ...

  private:
    double length_;
};
```

32

Encapsulation

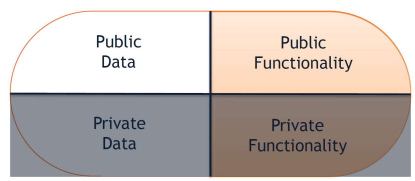
Encapsulation encloses data and functionality into a single unit (called a **class**):



33

Encapsulation

In C++, data and functionality are separated into two separate protections: **public** and **private**.



34

Public vs. Private

- The protection level determines the access that "client code" has to the member data or functionality:
- **Public** members can be accessed by client code.
- **Private** members cannot be accessed by client code (only used within the class itself).

35

Class syntax - Example

```
// A class for simulating an integer memory cell

class  IntCell
{
  public:
      IntCell( ){
        storedValue = 0;
      }

      IntCell(int initialValue ) {
        storedValue = initialValue;
      }

      int read( ) {
        return storedValue;
      }

      void write( int x ) {
        storedValue = x;
      }

  private:
      int storedValue;
};
```

constructors

36

Object declaration and use

- In C++, an object is declared just like a primitive type.

```
#include <iostream>
#include "IntCell.h"

using namespace std;

int main()
{
   //correct declarations
   IntCell m1;
   IntCell m2 (8);
   IntCell *m3;

   // program continues in the next slide
```


Object use in a client program

```
   // program continues
   m1.write(44);
   m2.write(m2.read() +1);
   cout << m1.read() << "   " <<  m2.read() << endl;
   m3 = new IntCell;
   cout << "m3 = " << m3->read() << endl;
   return 0;
}
```


Dynamic Memory Allocation

- **`new`** and **`delete`**
  - `new` - automatically creates object of proper size, calls constructor, returns pointer of the correct type
  - `delete` - destroys object and frees space
  - You can use them in a similar way to `malloc` and `free` in C.
- **Syntax**:
  - `TypeName *typeNamePtr;`
  - `typeNamePtr = new TypeName;`
    - new creates TypeName object, returns pointer (which typeNamePtr is set equal to)
  - `delete typeNamePtr;`
    - Calls destructor for TypeName object and frees memory

Examples

```
// declare a ptr to user-defined data type IntCell
IntCell *ptr1;

int *ptr2;

// dynamically allocate space for an IntCell;
// initialize values; return pointer and assign
// to ptr1
ptr1 = new IntCell(5);

// similar for int:
ptr2 = new int(2);

// free up the memory that ptr1 points to
delete ptr1;
```



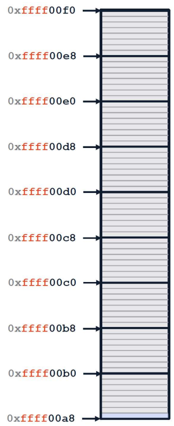
```
// dynamically allocate array of 23 IntCell slots
// in each storedValue will be initialized to 0
ptr1 = new IntCell[23];

// similar for int
ptr2 = new int[12];

// free up the dynamically allocated array
delete [] ptr1;
```


Stack Memory

- By default, every variable in C++ is placed in **stack memory.**
- Stack memory is associated with the current function and the memory's lifecycle is tied to the function:
  - When the function returns or ends, the stack memory of that function is released.

Stack Memory

- Stack memory always starts from high addresses and grows down:



43

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

void foo(){
   int x = 42;
   cout << " x in foo(): " << x << endl;
   cout << "&x in foo(): " << &x << endl;
}

int main(){
    int num = 7;
    cout << " num in main(): " << num  << endl;
    cout << "&num in main(): " << &num  << endl;
    foo();
    return 0;
}
```

```
 num in main(): 7
&num in main(): 0x6dfefc
 x in foo(): 42
&x in foo(): 0x6dfecc

Process returned 0 (0x0)   execution time : 0.075 s
Press any key to continue.
```

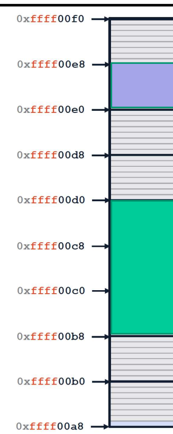
44

Stack Memory

- When a function returns, its stack memory is released.

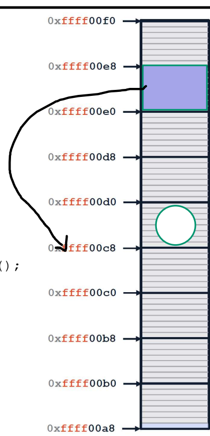
45

```
#include "IntCell.h"
using namespace std;

IntCell *CreateIntCell(){
   IntCell x;
   x.write(15);
   return &x;
}

int main(){
    IntCell *p = CreateIntCell();
    someOtherFunction();
    int a = p->read();
    return 0;
}
```



46

```
#include "IntCell.h"
using namespace std;

IntCell *CreateIntCell(){
   IntCell x;
   x.write(15);
   return &x;
}

int main(){
    IntCell *p = CreateIntCell();
    someOtherFunction();
    int a = p->read();
    return 0;
}
```



47

Heap Memory

- If memory needs to exist for longer than the lifecycle of the function, we must use **heap memory.**
  - The only way to create heap memory in C++ is with the **new** operator.
- The **new** operator returns a **pointer** to the memory storing the data – not an instance of the data itself.

48

C++'s `new` operator

- The `new` operator in C++ will always do three things:
  1. Allocate memory on the heap for the data structure
  2. Initialize the data structure
  3. Return a pointer to the start of the data structure
- The memory is only ever reclaimed by the system when the pointer is passed to the `delete` operator.

49

Heap Memory

The code below allocates two chunks of memory:

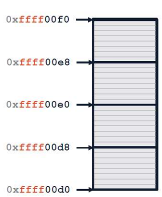
- Memory to store an **integer pointer** on the **stack**
- Memory to store an **integer** on the **heap**

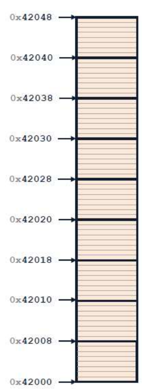
```
int *numPtr= new int;
```

50

Stack and Heap Memory

```
int main(){
   int *p = new int;
   IntCell *c = new IntCell;
   *p = 42;
   (*c).write(4);
         // or: c ->write(4);
   delete c;
   delete p;
   return 0;
}
```





51

`nullptr`

- The C++ keyword `nullptr` is a pointer that points to the memory address 0x**0**.
- `nullptr` represents a pointer to "nowhere"
- Address 0x0 is reserved and never used by the system.
- Address 0x0 will always generate an "segmentation fault" when accessed.
- Calls to `delete` 0x0 are ignored.

52