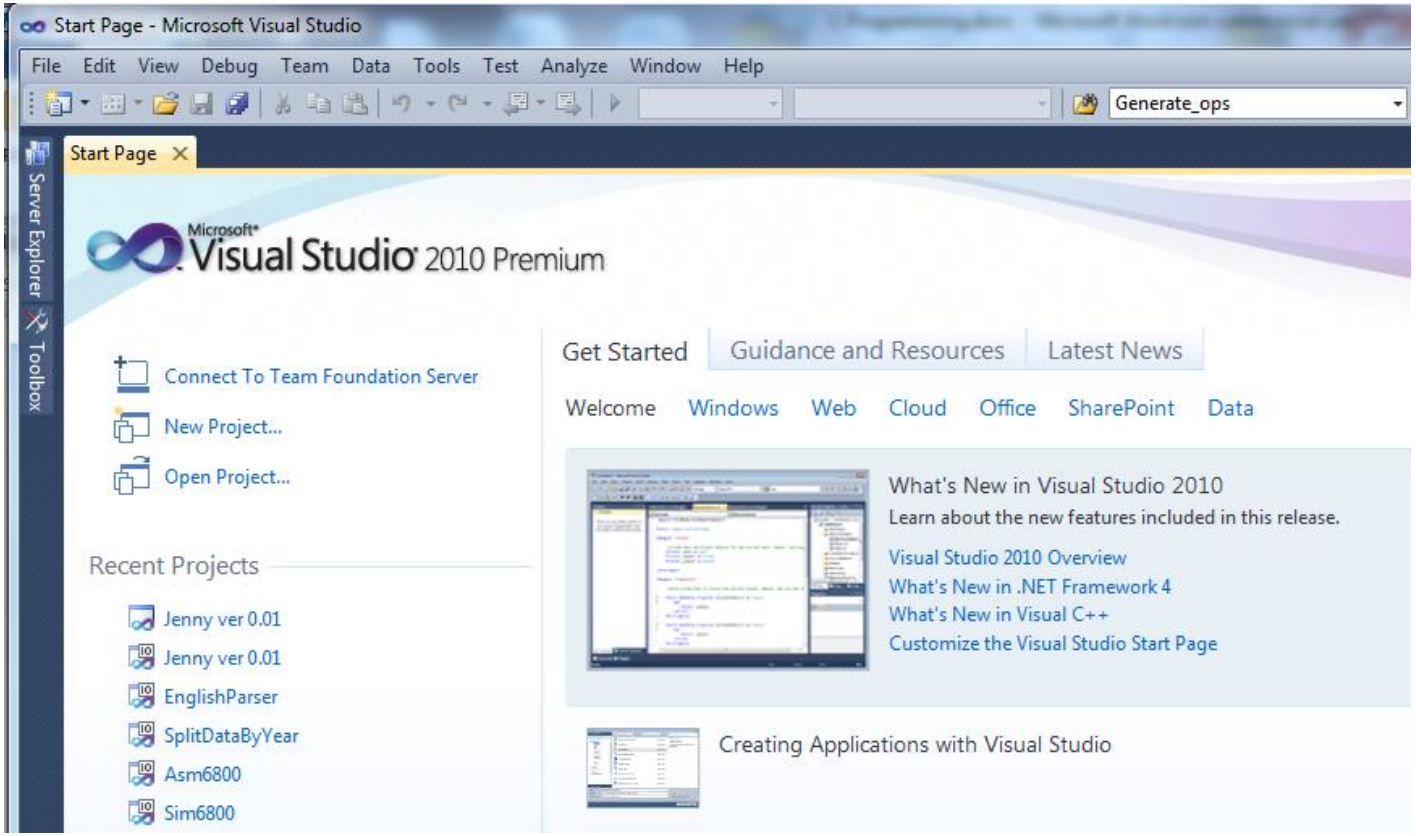
UFCF93-30-1 Computer and Network Systems

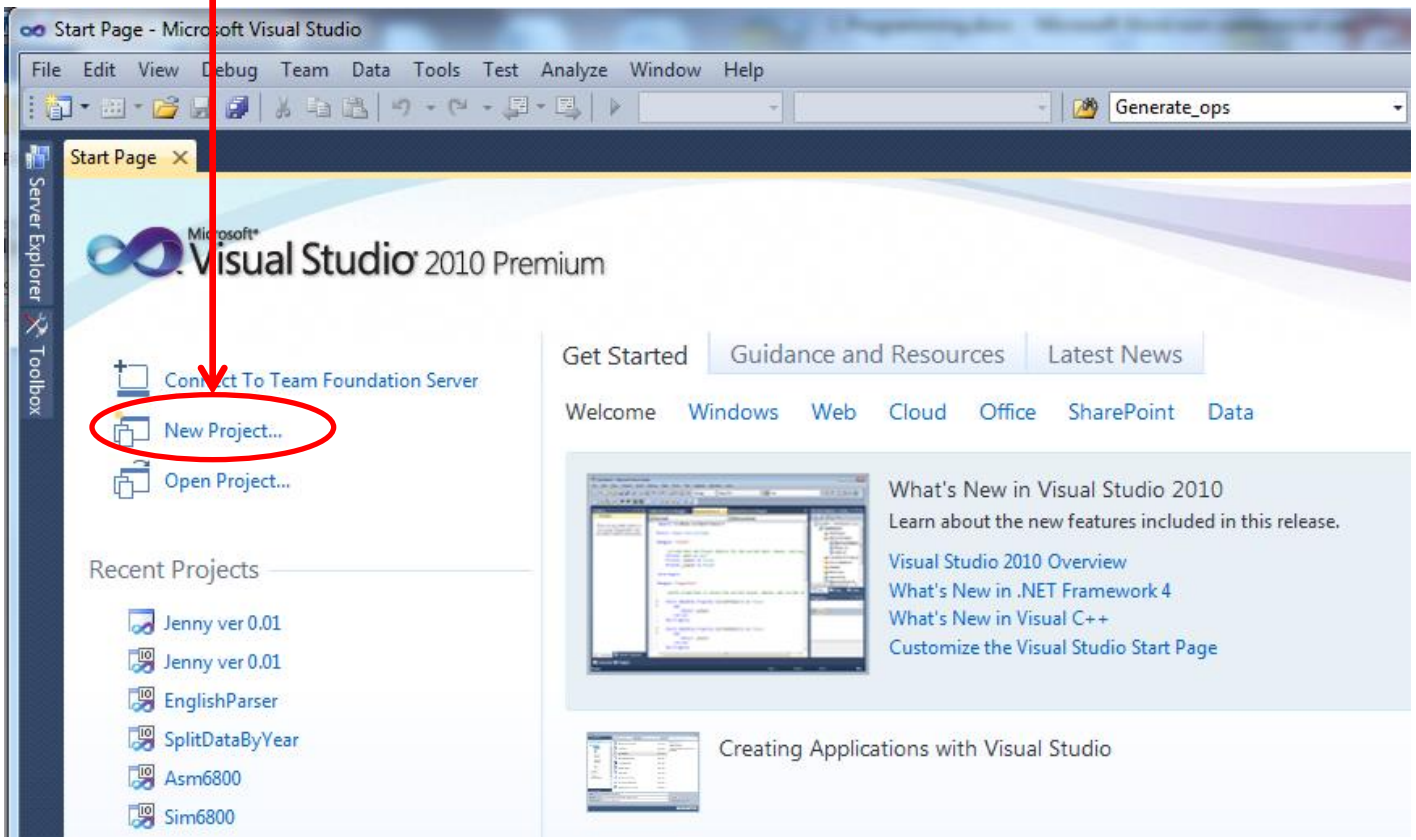
Computer Practical 1 Learning C programming

Start **Visual Studio** from the Program Menu, bottom left



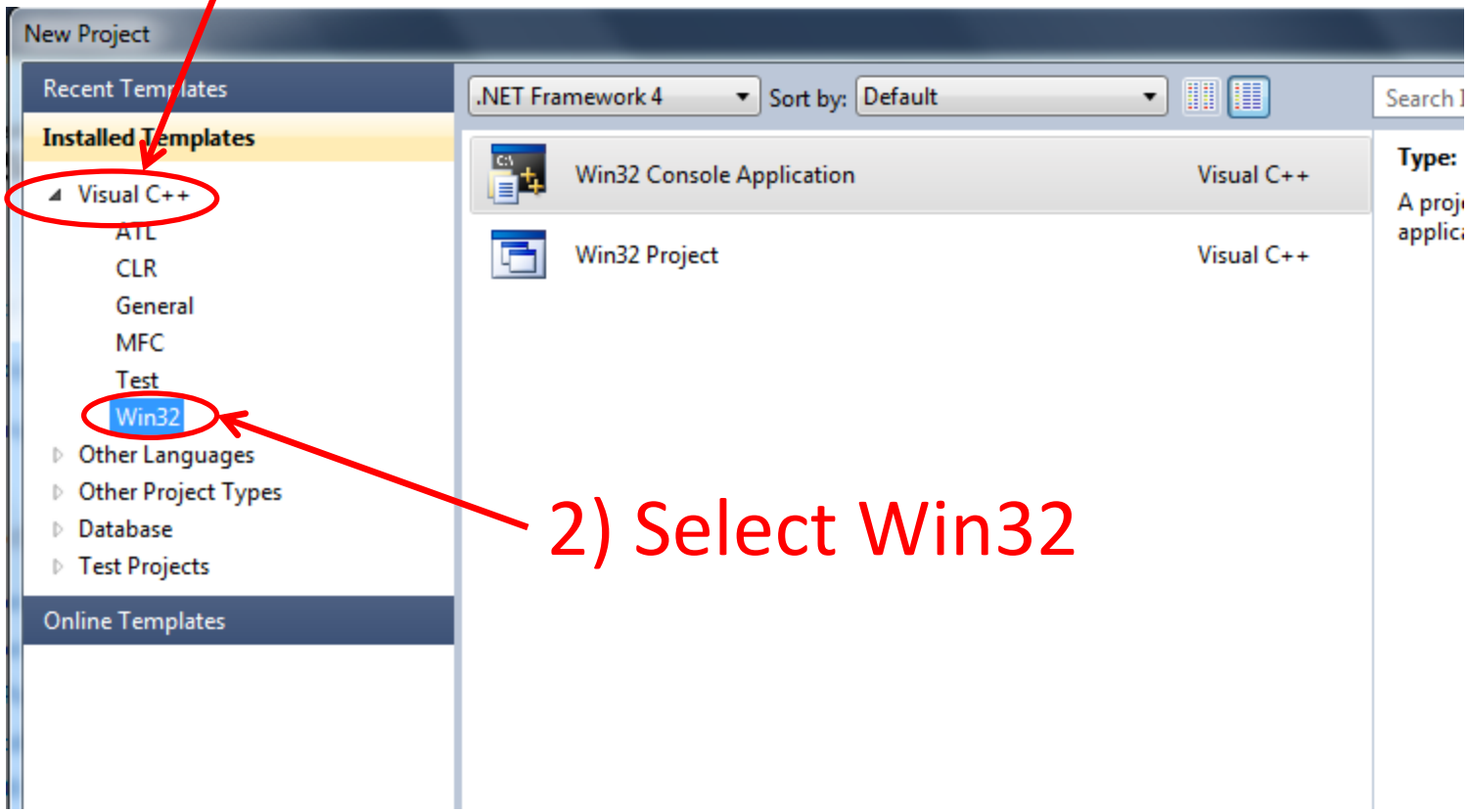
C Programming

1) Select New Project…



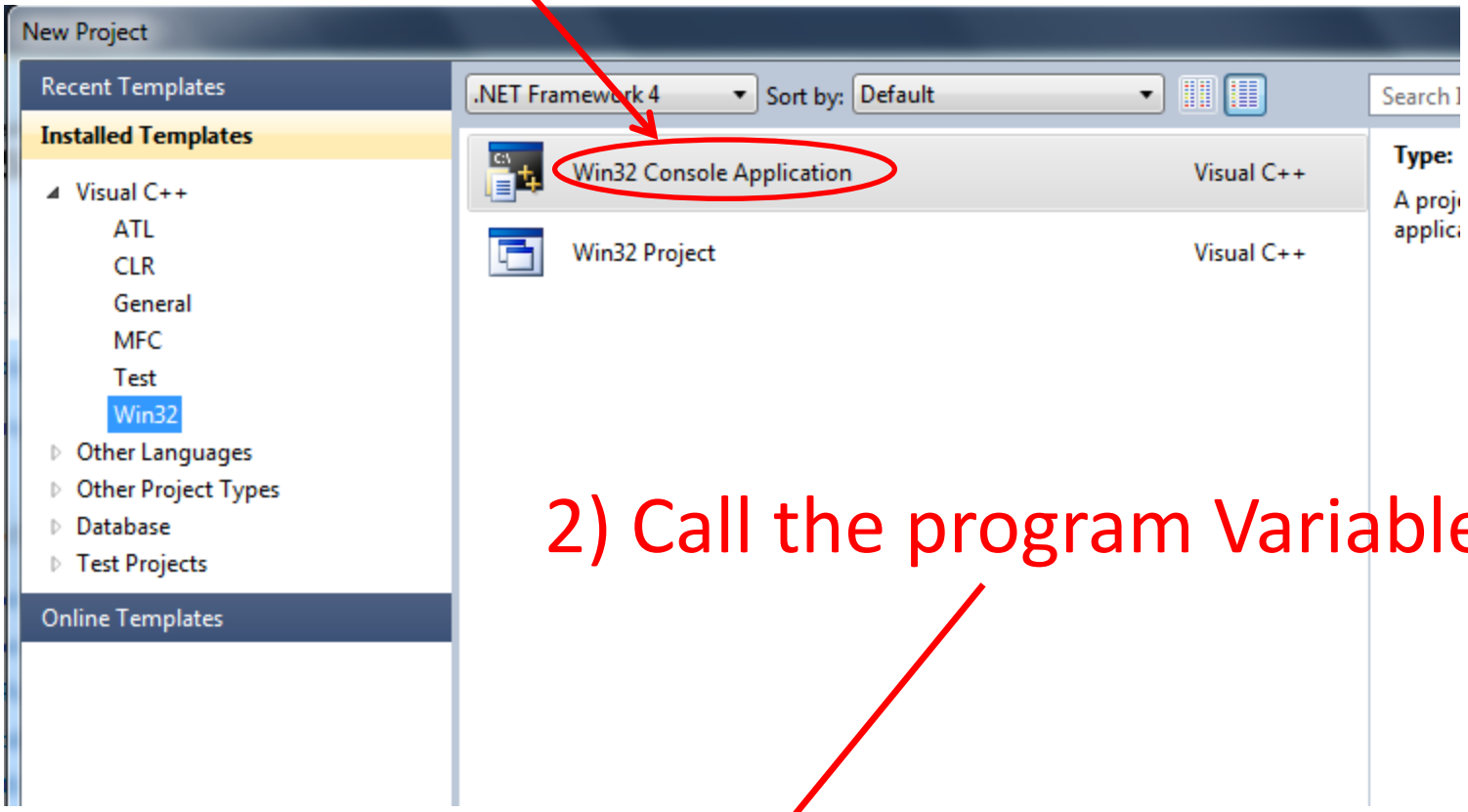
C Programming

1) Select Visual C++



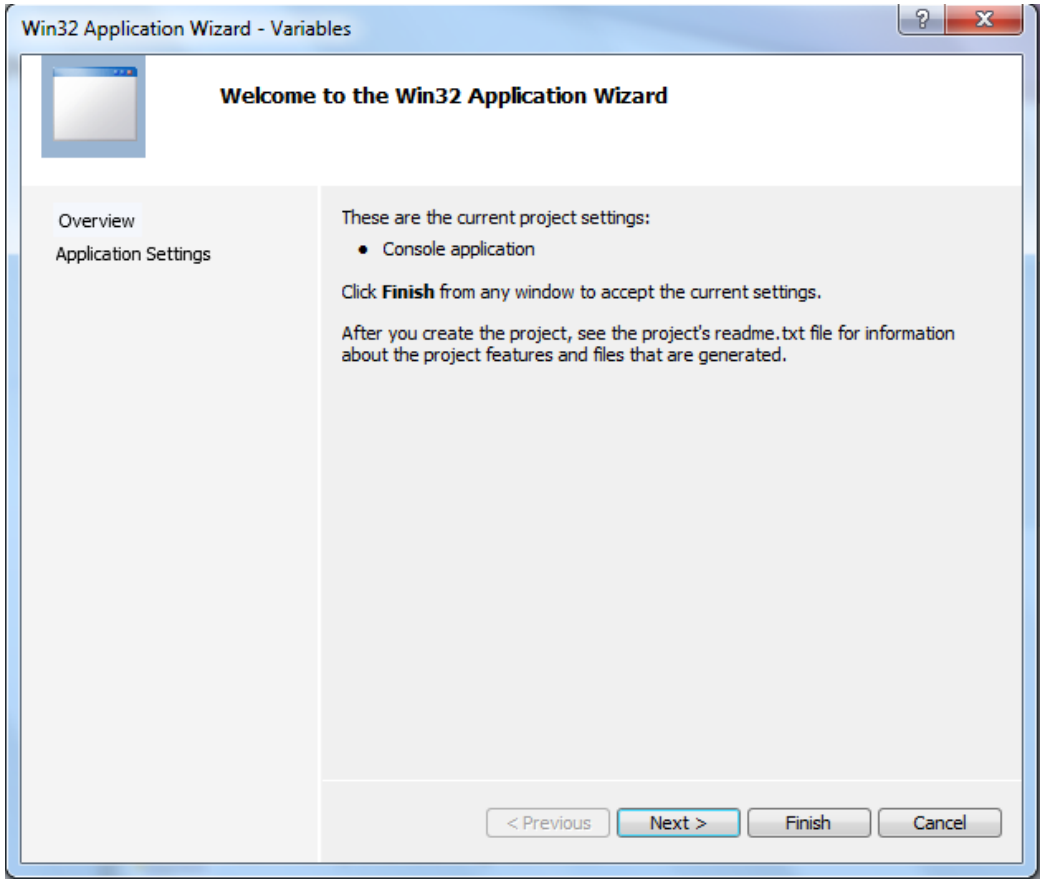
C Programming

1) Select Win32 Console Application

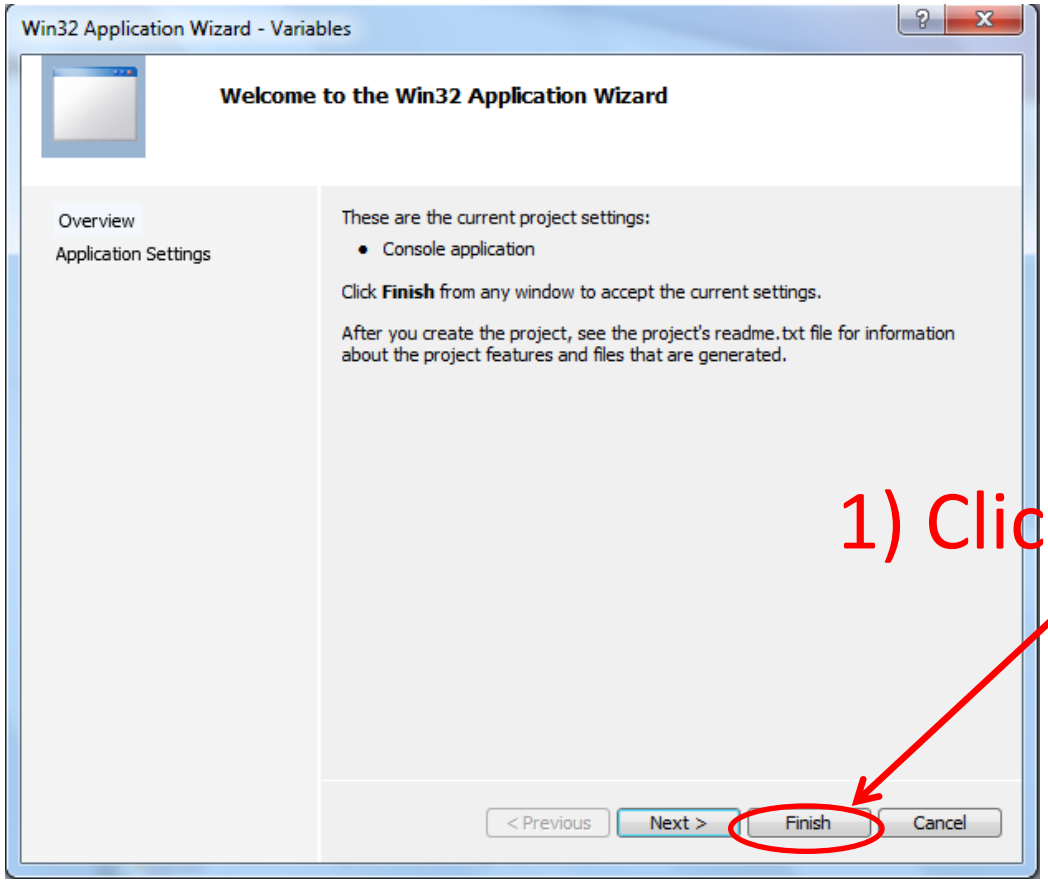


2) Call the program Variables

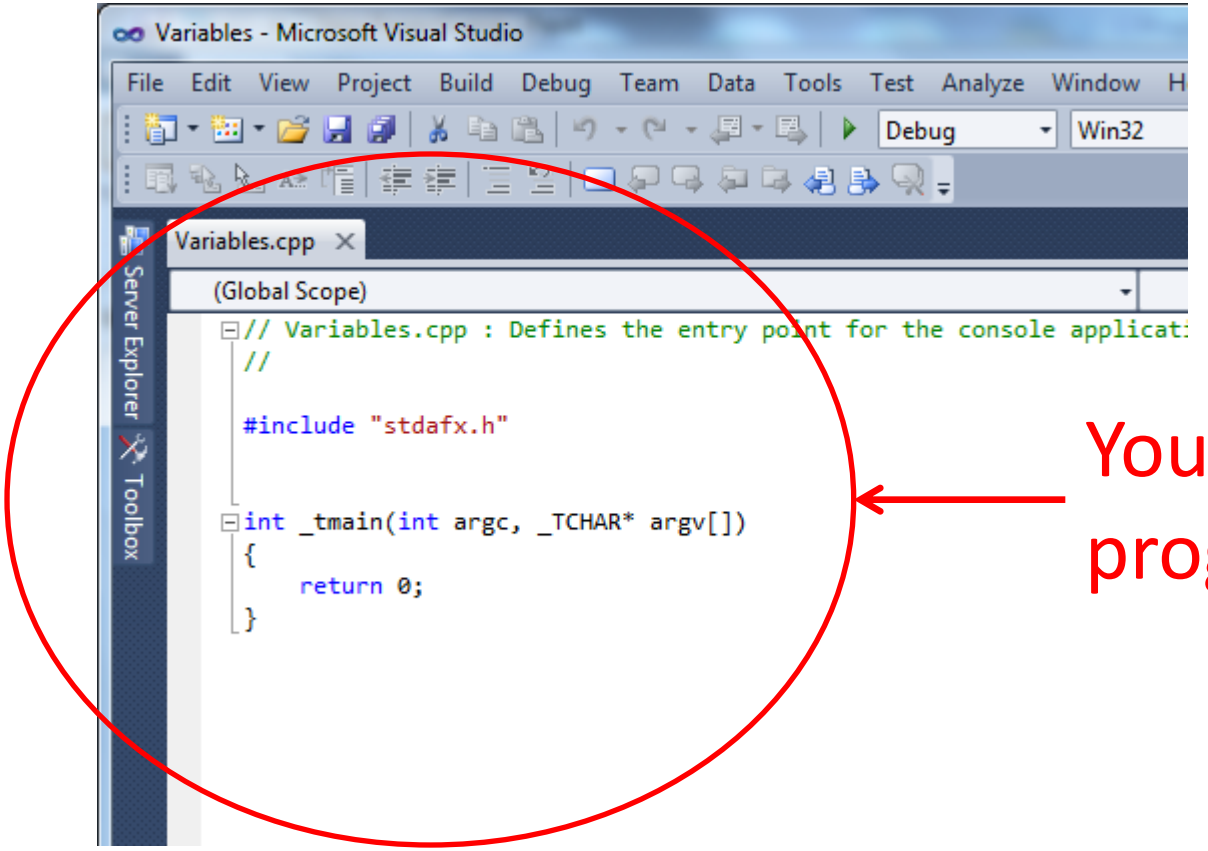
C Programming



C Programming



C Programming

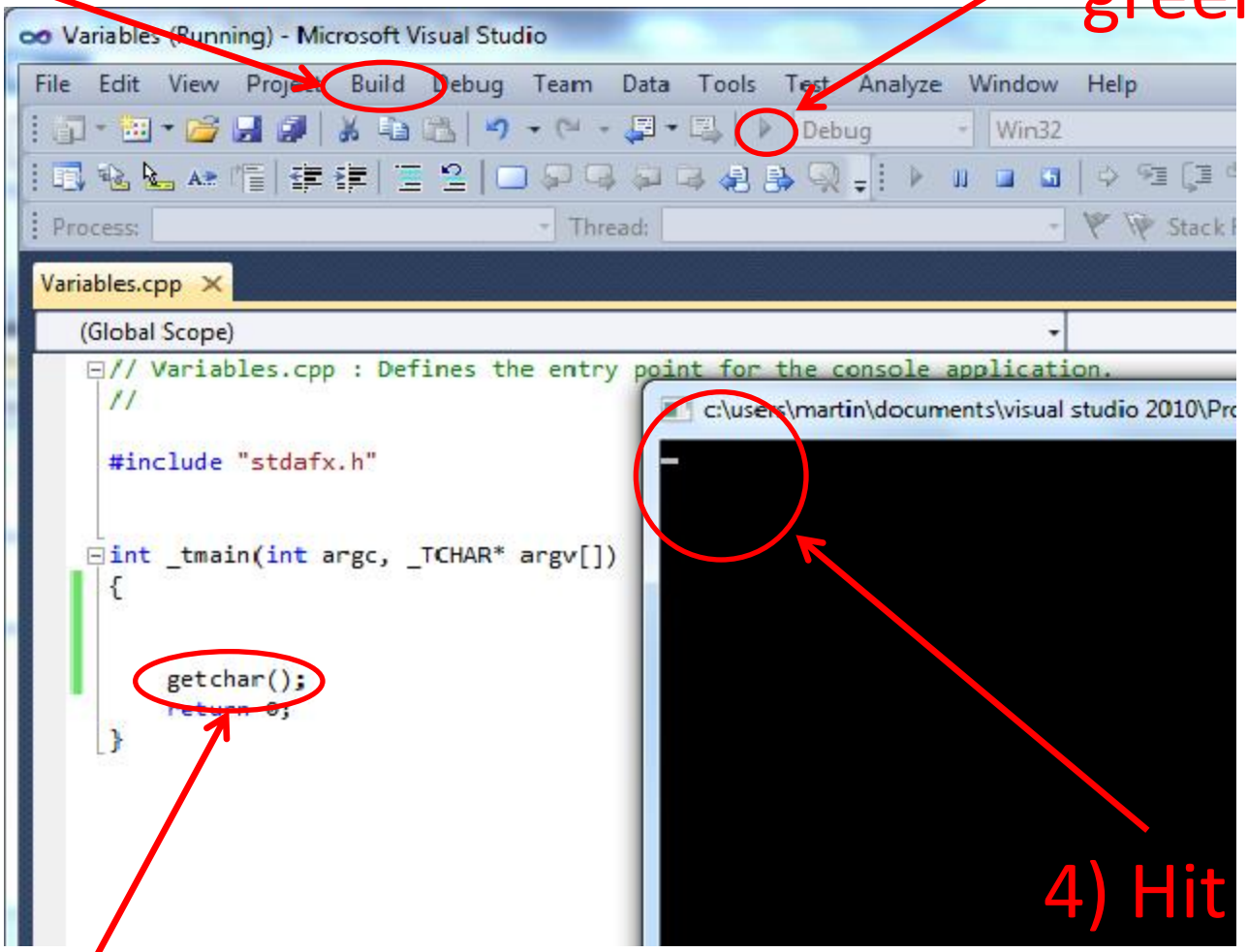


Your new program

C Programming

2) Click Build

3) Click the green arrow



1) Add getchar();

4) Hit any key to close the program

C Programming - integers

In C there are different types of variables
Each takes up a different size piece of memory

Variable	Size in bits
char	8
short	16
long	32
int	16 or 32
float	32
double	64

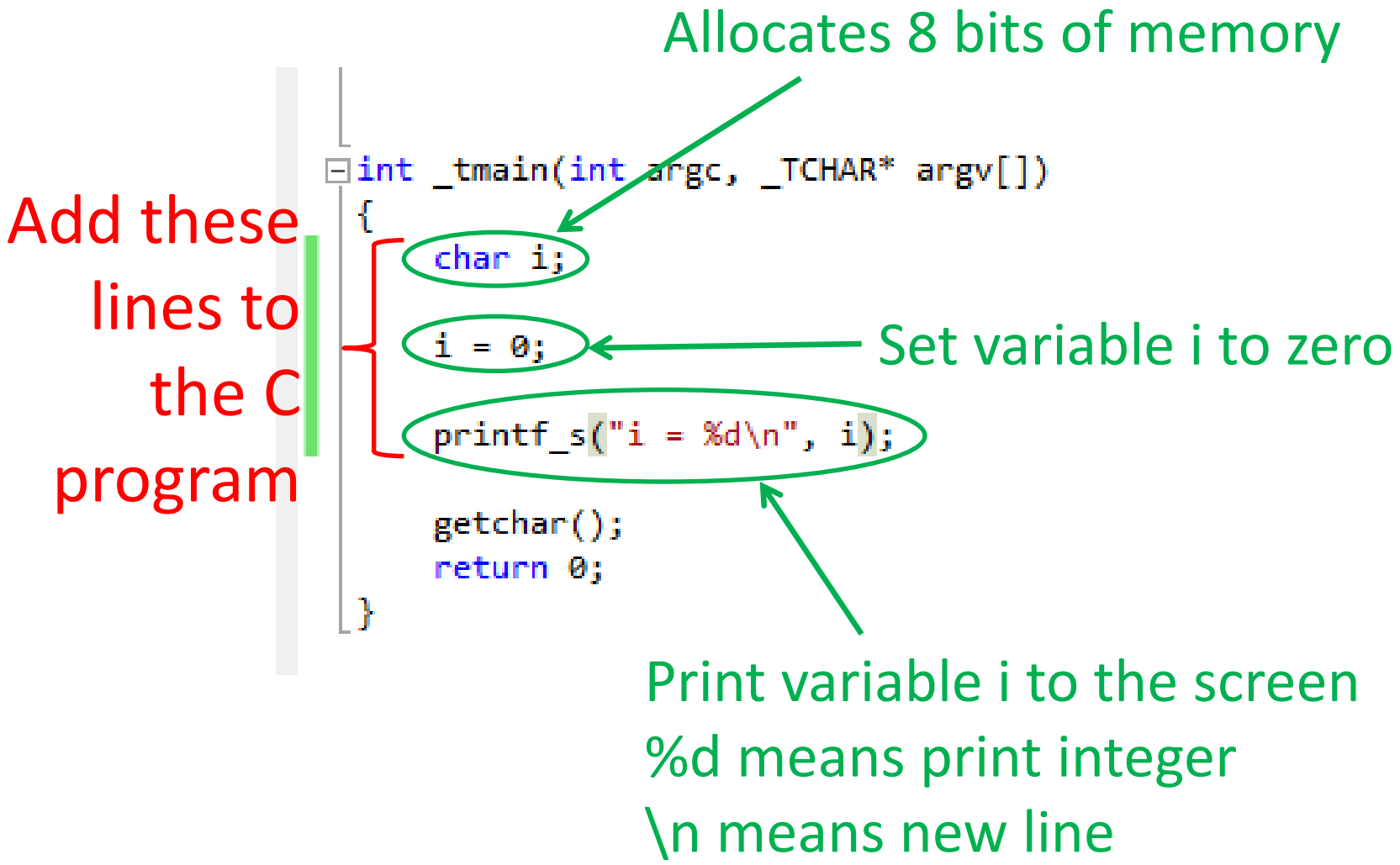
C Programming - integers

Lets look at **char**…

It is an **8-bit integer** that is usually used to store **ASCII characters**. An array of char is a string of text.

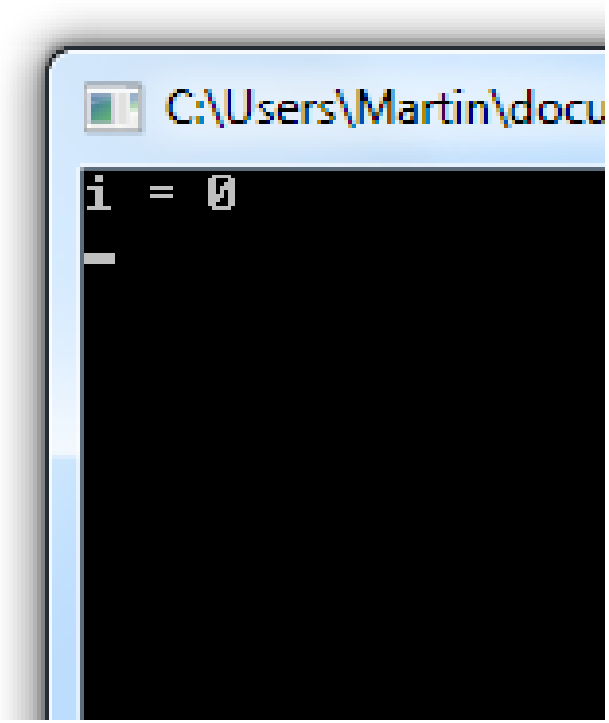
But for now lets treat it as a small integer
As a signed integer (default) it has a range **-128 to 127**

C Programming - integers



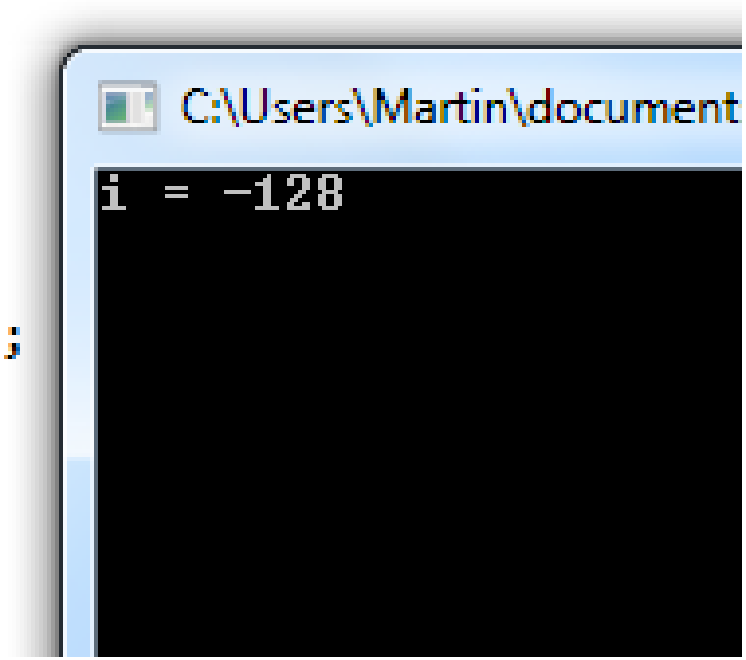
Build and run the program

```
int _tmain(int argc, _TCHAR* argv[])
{
    char i;

    i = 0;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```



C Programming - integers

```
int _tmain(int argc, _TCHAR* argv[])
{
    char i;

    i = -128;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```

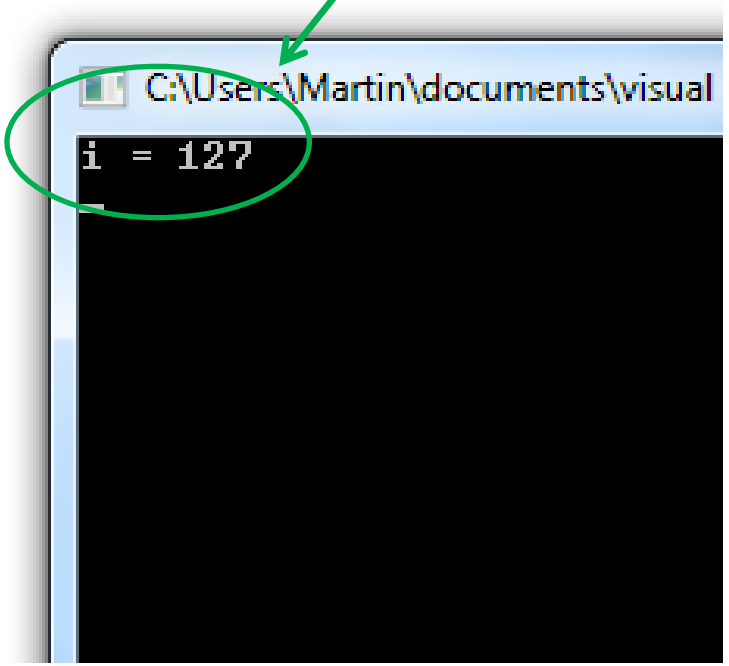


Set i to **-128** then build and run

C Programming - integers

What has happened here?

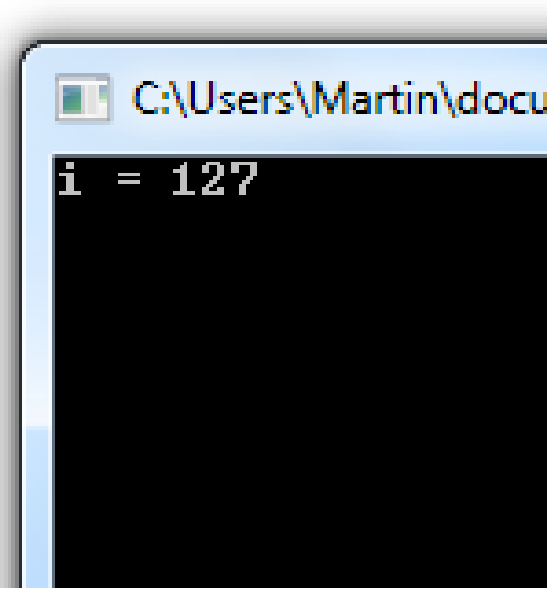
```
int _tmain(int argc, _TCHAR* argv[])
{
    char i;

    i = -128;

    i = i - 1;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```



Add a line to decrement i by 1 then build and run

C Programming - integers

```
int _tmain(int argc, _TCHAR* argv[])
{
    char i;

    i = 127;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```

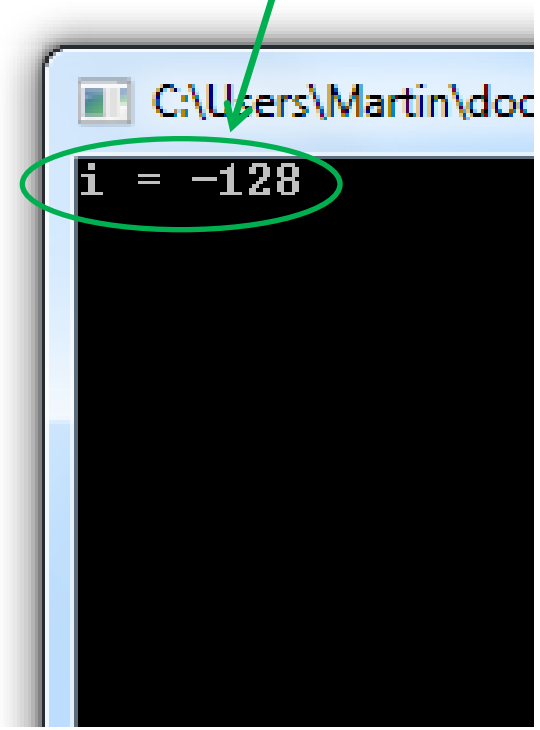


Set i to **127** then build and run

C Programming - integers

What has happened here?

```
int _tmain(int argc, _TCHAR* argv[])
{
    char i;

    i = 127;

    i = i + 1;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```

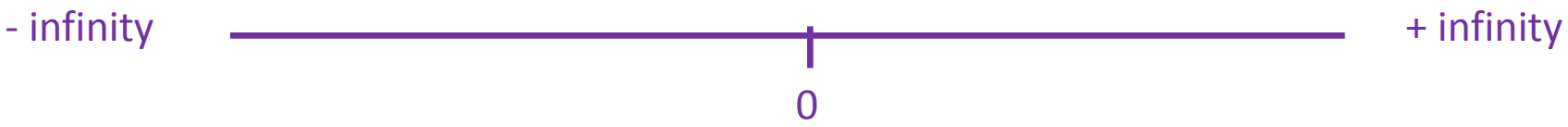


Add a line to increment i by 1 then build and run

So why is our char variable misbehaving?

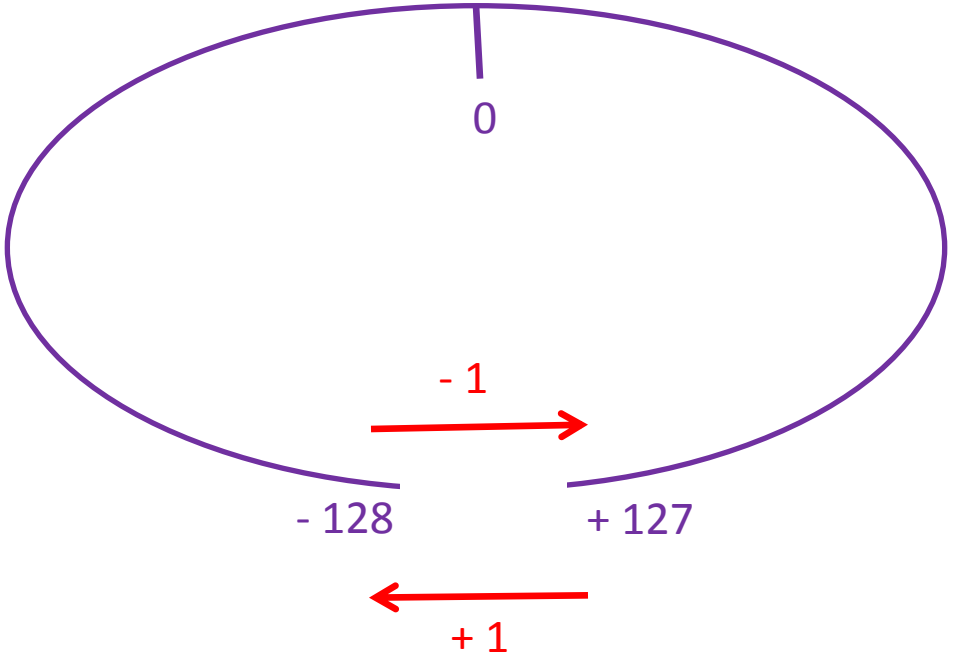
So why is our char variable misbehaving?

Normally when we think of signed integers they go from minus infinity to plus infinity



So why is our char variable misbehaving?

In the world of computing variables can wrap around!



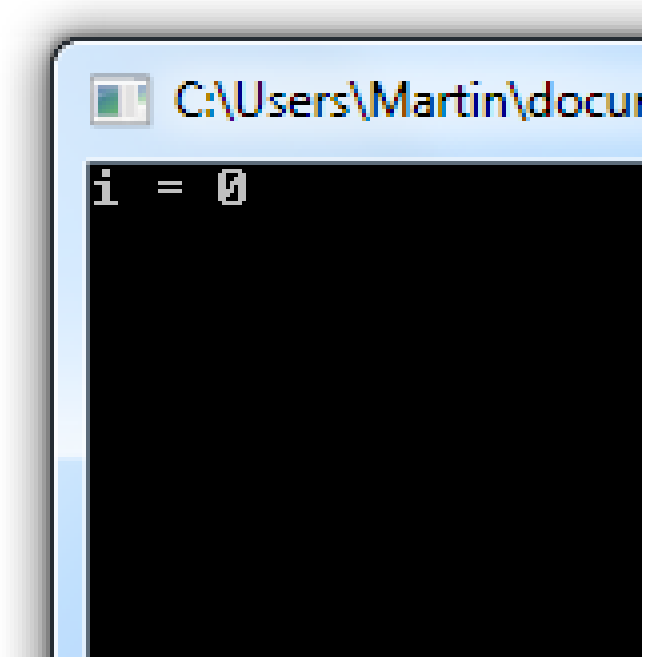
The same is true if we use unsigned integers

An **unsigned char** can store a value in the range of **0 to 255**

Try it!

C Programming - integers

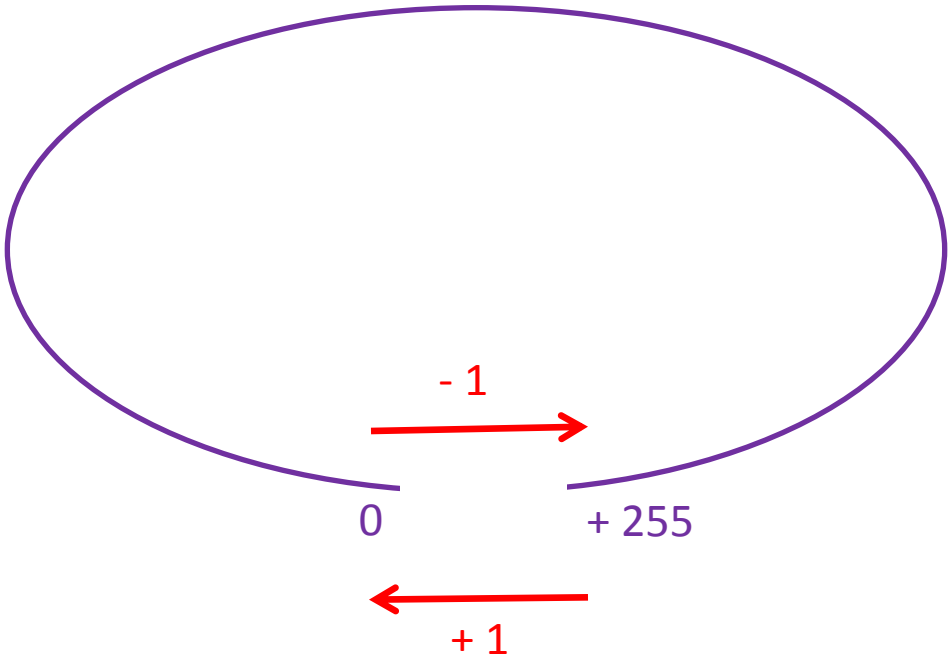
```c
int _tmain(int argc, _TCHAR* argv[])
{
    unsigned char i;

    i = 0;

    printf_s("i = %d\n", i);

    getchar();
    return 0;
}
```



C Programming - integers

Try…

1) Subtract 1 from 0
2) Add 1 to 255



C Programming - integers

The same is true for all integer types… the range just gets bigger

Type	Size in bits	From	To
char	8	-128	127
unsigned char	8	0	255
short	16	-32768	32767
unsigned short	16	0	65535
long	32	-2147483648	2147483647
unsigned long	32	0	4294967295

So you always need to be aware of the range of values that can be stored

Programmers that forget can cause some nasty problems!

C Programming – floating point

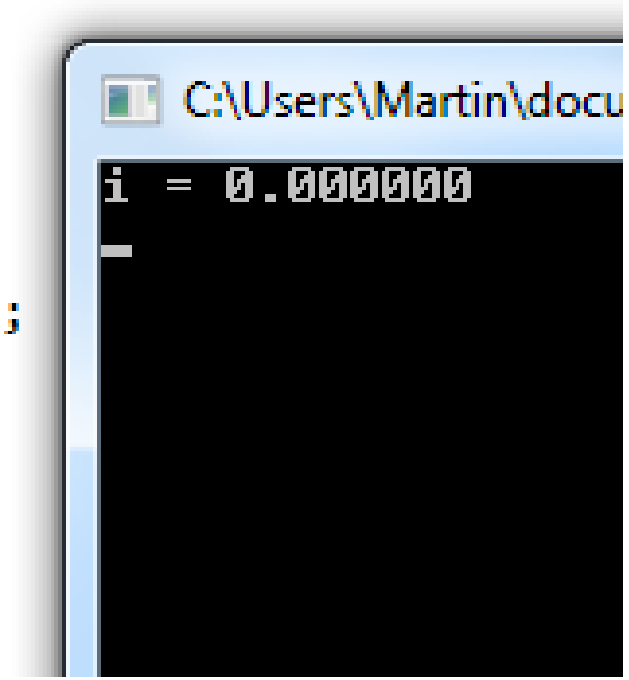
We have seen with integers that you need to be careful with the range of values that can be stored within a certain size integer

With **floating point** numbers we have a **problem of range** but more importantly we have a **problem of precision**

C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float i;

    i = 0.0;

    printf_s("i = %f\n", i);

    getchar();
    return 0;
}
```

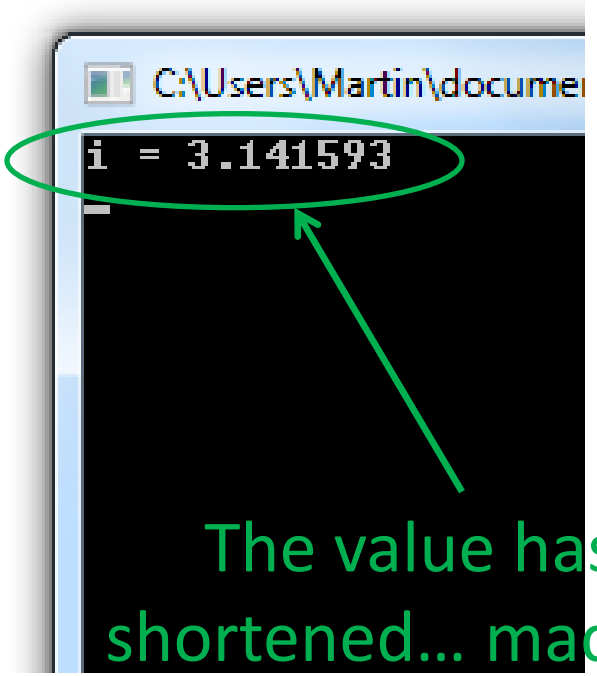
Add these lines to your program



%f means print out a floating point number

C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float i;

    i = 3.14159265359;

    printf_s("i = %f\n", i);

    getchar();
    return 0;
}
```



Change i to 3.14159265359 then build and run

The value has been shortened… made less precise as the full precision cannot be stored

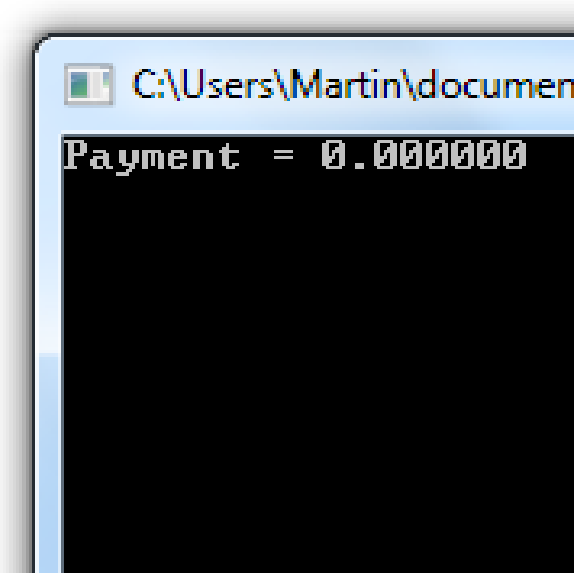
C Programming – floating point

Lets have some more fun with floating point numbers…. Imagine that you are responsible for writing a **system for a bank**

C Programming – floating point

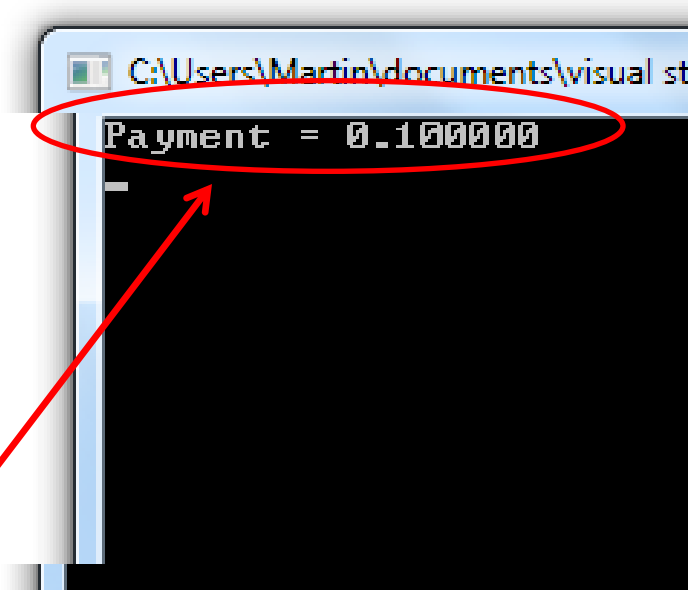
Lets create a payment system…

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;

    payment = 0.0;

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```



C Programming – floating point

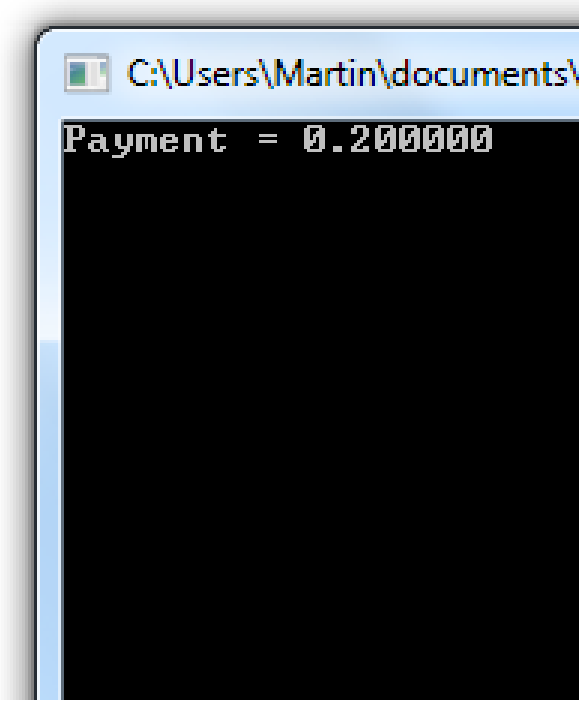
```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;

    payment = 0.0;

    payment = payment + 0.1;

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```



Lets make a payment of £0.1 (i.e. 10p)

C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;

    payment = 0.0;

    payment = payment + 0.1;
    payment = payment + 0.1;

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```



Lets make two payments of £0.1 (i.e. 10p). A total of 20p

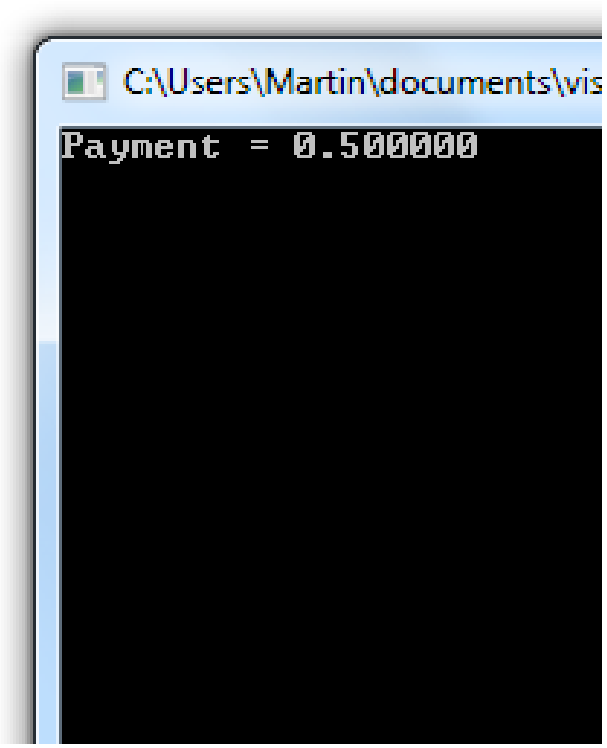
C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;
    int i;

    payment = 0.0;

    for (i=0; i<5; i++)
    {
        payment = payment + 0.1;
    }

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```

Initialise loop counter i to zero

Go around loop while i is less than five

Increment loop counter i at the end of the loop



This time we make five payments of £0.1 (10p), but this time we do it using a **for** loop, 50p in total.

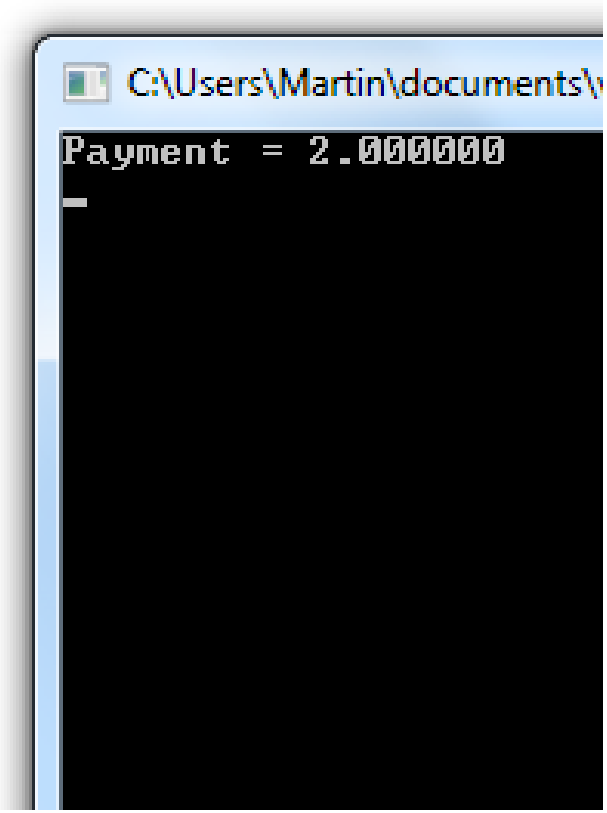
C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;
    int i;

    payment = 0.0;

    i=0;
    while ( i<5 )
    {
        payment = payment + 0.1;

        i++;
    }

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```

The same but using a while loop

C Programming – floating point

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;
    int i;

    payment = 0.0;

    for (i=0; i<20; i++)
    {
        payment = payment + 0.1;
    }

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```

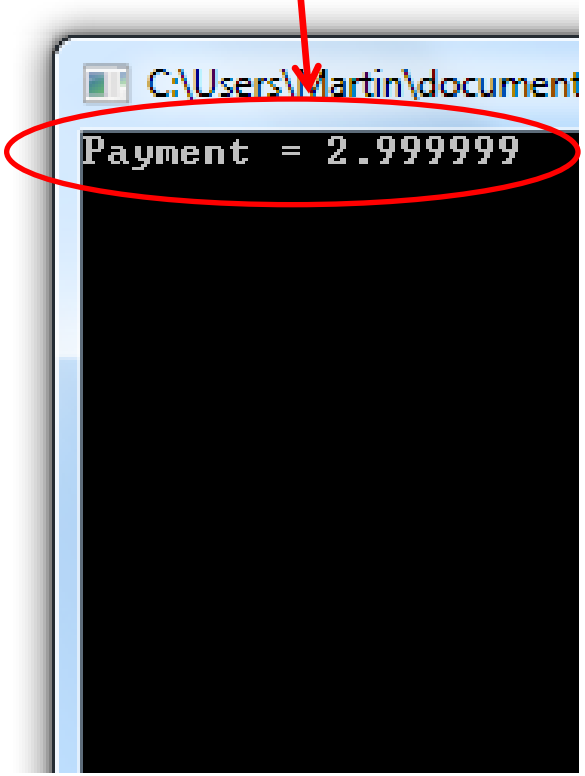


This time we make twenty payments of £0.1 (10p), £2 in total.

C Programming – floating point

But what has gone wrong???

```
int _tmain(int argc, _TCHAR* argv[])
{
    float payment;
    int i;

    payment = 0.0;

    for (i=0; i<30; i++)
    {
        payment = payment + 0.1;
    }

    printf_s("Payment = %f\n", payment);

    getchar();
    return 0;
}
```



This time we make thirty payments of £0.1 (10p), £3 in total.

C Programming – floating point

All numbers are stored as bits. Each bit can be 0 or 1. Because of this everything is in base 2. So when a computer stores a fraction it stores it as combinations of ½ and ¼ and 1/8 and 1/16 etc.

Money is in base 10. A tenth cannot exactly be made up from a combination of ½ and ¼ and 1/8 and 1/16 etc.

C Programming – floating point

So, if you are writing a banking system do not represent money using floating point numbers

Instead you should use integers to represent pennies… just make sure you have used large enough integers to store all the money else someone's bank account might go to negative when you add money to it

The End